VOLUME 05 | ISSUE 12

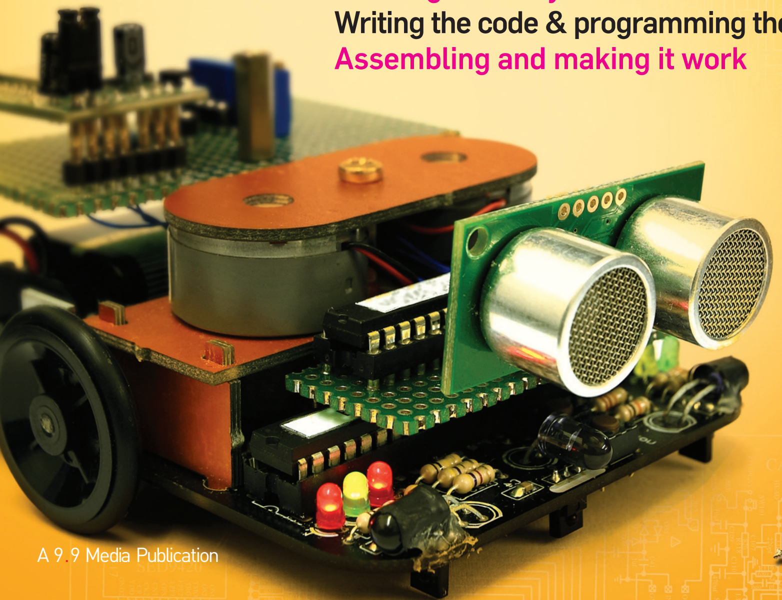
digit FastTrack

YOUR HANDY GUIDE TO EVERYDAY TECHNOLOGY

# to BUILD YOUR OWN ROBOT

Introduction to Robotics
Introduction to Electronics
Introduction to Mechanics
Introduction to Programming
Algorithm and Logic
Designing the circuit
Making the circuit
Building the body
Writing the code & programming the PIC
Assembling and making it work

DIY: BUILD YOUR OWN ROBOT



A 9.9 Media Publication

digit

YOUR TECHNOLOGY NAVIGATOR

thinkdigit.com

Fast Track

to

# Build your own Robot

thinkdigit.com

# Credits

The People Behind This Book

Editorial

|  |  |
|---|---|
| Editor | Robert Sovereign-Smith |
| Head-Copy Desk | Nash David |
| Writer | Puneet Garg |

Design and Layout

|  |  |
|---|---|
| Lead Designer | Vijay Padaya |
| Senior Designer | Baiju NV |
| Cover Design | Anoop PC |

# Contents

# Introduction

Robots and humanoids have always ruled our imagination. Last month, we covered the FIRA event in Bangalore and were excited to see how robots battled in a game of soccer. Besides, it's exciting to see how a machine behaves when it is endowed with logic to decide, interpret and react to its surroundings. In fact, it's difficult to find someone who is not fascinated by "intelligent" machines. Machines that can walk, talk and see invokes the kid in each one of us. Robots have always attracted hobbyists and professionals alike.

There has always been a dearth of good books that teach you how to build a robot yourself. Pick up any book on robotics, and soon you'll be overwhelmed with the daunting mathematical equations, dreadful diagrams and a feeling that building a robot is too technical and difficult.

We're presenting you with a step-by-step guide to build basic robots, without going into those tough mathematical and technical details. So no matter what your background is, this book is self-sufficient to guide you in making your first robot.

We will start by understanding the basics of robot designing. We will build a fully autonomous robot which will follow a line on its own. There are individual chapters fully dedicated to each aspect of building an autonomous robot; which are mechanics, electronics and programming. Each chapter proceeds in a very lucid way.

Throughout the process of building this robot, we will be working with 12 V DC, so you need not worry about any electrical hazards. The best way to learn something is dive in! We suggest you to actually build a robot along, as you read. Also, don't miss the associated video in this month's DVD.

Chapter 1 begins with an Introduction to robotics. This is aimed at getting you familiar with the different kinds of robots for a better understanding of this exciting field.

Chapter 2 talks about electronics. This is meant for those of you who are not familiar with electronics or electronic components and terminologies.

Chapter 3 talks about the mechanical parts associated with Robotics. Here you'll get familiar with the various parts you will be using.

Chapters 4 and 5 talk about programming concepts and code in general. We also suggest you watch the associated video on the DVD for a more detailed explanation of the terms and concepts in this chapter.

Chapters 6, 7 and 8 takes you a step closer to your robot by talking about designing the circuit for your robot.

Finally the Fast Track concludes with the necessary steps you need to follow to tweak your robot in case of any malfunction. You will find the necessary directions from the DVD in the form of an exclusive video that we've included. We hope you will find your new hobby extremely exciting. Do write in to us with your comments and pictures of your finished robot. Alternatively, you can also share your experiences on our Facebook page at **www.thinkdigit.com/facebook**.

# 1 Introduction to robotics

Although robotics is a vast field, broadly, robots are of three types – operator controlled, autonomous, and semi-autonomous.

## 1.1 Operator controlled

These robots are remotely controlled by a human. They do not require any sensor or processing unit as the human performs these functions and accordingly steers the robot.

## 1.2 Autonomous

These robots do not need any operator. They have sensors and processors in addition to actuators (motors). They sense the state around the robot, process the information and act accordingly. This category is, undoubtedly, the most fascinating category of robots as autonomous robots resemble living organisms and watching them move and doing things on their own, is a pleasure for hobbyists and robotics enthusiasts.

## 1.3 Semi-autonomous

These are a hybrid of autonomous and operator-controlled robots. They may or may not need sensors. Usually, they have a processor, which works in sync with the operator, and instructs the actuators accordingly. The process of making a robot requires a process-driven approach.

You need to have a complete plan of action before you begin building your robot. First, you need to study about various aspects of robot building. Next, you need to know the various methods of building the body. After that, you need to be familiar with the logic you would be implementing, and also designing the circuit. This is called 'brainstorming'. After brainstorming, explore available resources and the best way to utilise them. The next step is designing. This involves designing the code, body and the circuit for your robot. Finally, comes the building and assembly stages of the project. If you missed out somewhere, you need to troubleshoot. The whole process would become tedious if you don't follow a sequential approach. So, it's always better to be systematic and follow the procedure we've mentioned here, especially if you're building a robot for the first time (which we assume to be the case). To sum it up, read this book from the first page to the last, rather than jumping straight to the end.

# 2 Introduction to electronics

With a basic knowledge of robotics and the various aspects of putting a robot together, it's now time to gain a basic understanding of electronic components and how circuits work.

## 2.1 Breadboards and PCBs

These are most commonly used for building circuits. PCBs stand for printed circuit boards. Rather than using connecting wire and component terminals to form a closed circuit, it is more efficient to use a firm board with a copper coating. The required conductors are drawn by removing copper from the board with a process called etching using a corrosive solution of Ferric Chloride ($FeCl_3$).

### Breadboards

Instead of PCBs, you could also use breadboards. With breadboards, you don't have to touch the soldering iron. All you need to do is plug component terminals in and out of the board to make or break connections. Breadboards have a lot of holes, connected internally into groups. You just have to insert your component's terminal into these holes – it's as simple as that.

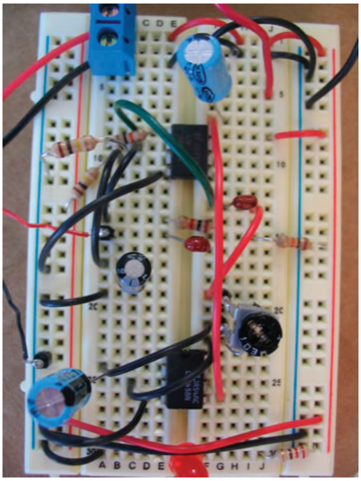
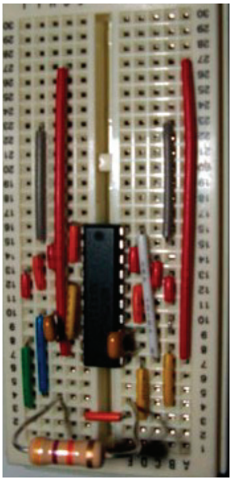
### PCB (Printed circuit board)

The motherboard of your PC is a standard example of a PCB. The use of breadboards has its own advantages over the use of PCBs. In addition to not having to solder at all, you could proceed onto building circuits once you've designed them. Also, you can reuse your components once you are done and save a lot of money.

On the other hand, once you solder the components and make the necessary connections on a PCB, they have longer shelf lives, and would sustain jerks and shocks. Therefore, ideally, circuits are first built on a breadboard, tested, remade and the final design is soldered on a PCB for practical use. Try following the two golden rules to make fool proof circuits on a breadboard:

Use single strand wires (wires with single conductor wires within the insulator coating) instead of wires with multiple strands (fine multiple conductor wires). Use wires having lengths as close as possible to the distance between the two holes to be connected.

Still, if you have doubt of connection between two holes, insert a safety pin in each of these holes and check the continuity between the safety pins (the probes of a multimeter are too thick to go into any of the holes).



Follow some basic discipline while laying the components on the breadboard

**Possible problems with breadboards**

There are some bread boards whose holes are too small, and you can't insert leads of voltage regulators such as 7805 into it, as its legs are slightly thicker. So be careful when you buy a new breadboard, and if possible, before buying them, check by inserting a 7805 into it.

Even if you buy a new breadboard, some sections of the board may have faulty internal connections. As a result, even a properly assembled circuit may not work as desired. Also, with passage of time, the internal connections of breadboard may get loose. So, it is important to buy a good quality breadboard.

## 2.2 A look at components

### 2.2.1 Resistor

Resistors are electronic components that offer resistance to the flow of an electric current through them. The magnitude of resistance offered is measured in 'ohms'. The higher the ohmic value of a resistor, the more resistance it offers to the flow of current.

One of the precautions you should take while using a resistor is not to bend its terminals too close to its body

Generally, along with the ohmic value of a resistor, the power rating of resistors is also used to classify them. The resistors available in the market have resistances in range of 1 ohm to 64 mega ohm and the power rating varies from 1/8 W to 20W. However if you want, you could get resistors of ratings outside this range also. For basic electronic purposes, resistors having power rating of ¼ W would suffice.

To measure the value of resistance of a given resistor, you could use two methods. First one is to read the colour code given on the value of resistance. The other method, and also the easier and faster one is to use a multimeter.

## 2.2.2 Potentiometer

The resistors we have discussed above have constant resistances, which do not vary unless they are heated too much. A potentiometer is also a resistor, but it has variable resistor which can be changed at will by the user. Potentiometer is rated according to a range of resistances which can be achieved by using it, along with its power rating. A potentiometer is very useful in applications in which you are not sure of the exact value of the resistance you would require. When only the range required is known, a potentiometer is used and then its resistance is varied so as to get desired output from the circuit.

The resistance of the potentiometer is varied by rotating the 'shaft' or 'knob' located on its body. Rotating it in one direction increases the resistance, while rotating in opposite direction decreases it.

A potentiometer has three terminals. The resistance between the extreme two terminals is constant, whereas the resistance between either of the end terminals and the middle terminal is variable.

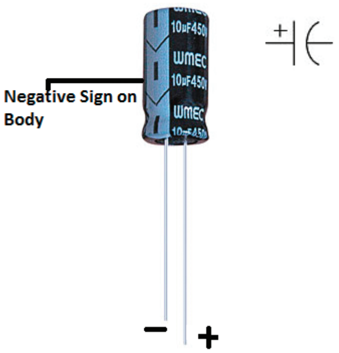
## 2.2.3 Capacitor

A capacitor is a device used to store electric charge. Their ability to store charge is measured in 'Farads'. The more the 'Farad' rating of the capacitor, the more charge it can store. But, Farad is a very large unit, so we commonly use smaller unit such as 'microfarads' to denote the capacitance of a capacitor.

### Polar Capacitors

These capacitors have fixed positive and negative terminals. These capacitors have to be connected in the correct way in the circuit. If you connect it in reverse polarity (connecting positive terminal where negative is to be connected and vice versa), an explosion may occur and you may spoil quite a few electronic components of your circuit.

The most widely available polar capacitors are electrolytic capacitors. You can identify the polarity of the terminals of an electrolytic capacitor by comparing their lengths. The longer terminal is always positive. Also, on the body of the capacitor, there is a negative sign mentioned above the negative terminal.



Electrolytic capacitors are polarised

### Non Polar Capacitors

These capacitors do not have any polarity and can be connected in either way in a circuit. Most commonly used non polar capacitors are mica and ceramic capacitors. Also, because both the terminals of the non-polar capacitor are identical, the symbol for them is slightly different than that for polar capacitors.

### Identifying capacitance values

A polar capacitor has both the capacitance value and the voltage rating clearly written on its body. However, for mica capacitors, they are written on the body in encoded form, generally a three digit number. If the number written on body is 474, then the value is 47 x 104. That is, the first two digits are written as they are, and the last digit is considered to be power of 10. Please note that the capacitances when decoded, give value in pico farads (pf). So 474 is equal to 47 x 104 pf and 342 is equal to 34 x 102 pf.

A ceramic capacitor and Symbol of a non polar capacitor

### 2.2.4 Multimeter

In both robotics and electronics, you need to measure three things very frequently:

#### Continuity between two points

To measure continuity, you should set the multimeter knob to continuity. Then, insert or place the probes of the multimeter in between the points to be checked for continuity. If you hear a beep, that means there is a connection between those two points.

#### Resistance between two points

Turn the knob to point towards the resistance measurement section and select upper value of the resistance randomly. Connect the probes across the points you want to measure the resistance. The resistance is displayed in ohms. If you see a 1, it means that the upper value (or limit) you selected is not appropriate, and the value of the resistance being measured is greater than it. In that case, you need to select a higher upper limit.

For example, if you want to measure a resistance and you set the knob to 20 kilo ohm. If the value of resistance is smaller than 20 kilo ohm, the multimeter screen will give you the value. But if the value of the resistance to be measured is greater than 20 kilo ohm, the multimeter will show 1, and in that case you need to select the 200 kilo ohm limit.

You need a multimeter through the building and troubleshooting stages

#### Voltage difference between two points

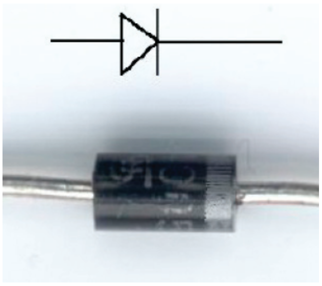
To check DC voltage difference between two points, set the knob to point in the direction of voltage measurement and hold the probes across the points you want to measure voltage. The upper limit is selected like in the case of resistance.

If the screen shows a voltage value along with a negative sign, it means the polarity is reversed. Interchange the red and black cables and the polarity will be positive.

And all the aforementioned quantities can easily be measured using a multimeter. The cost of a multimeter increases with the number of quantities it can measure. However, for basic electronics and robotics purpose a simple and cheap multimeter would suffice. It will cost around ₹100. A snapshot of such a cheap and easy to use multimeter is given alongside. Also, the sections we would need to use frequently are highlighted.

## 2.2.5 Diodes

Diodes are two terminal devices just like the resistors, but they conduct electricity only in one direction. A resistor offers same resistance to the flow of electricity irrespective of its direction through it. On the other hand, a diode offers very large resistance to flow of electricity through it in one direction, and negligible resistance in other direction. Essentially, it means that a diode behaves as an open circuit (allowing no current flow) in one direction and as a closed circuit (allowing all the current to pass through it) in other direction. The condition of maximum current flow (or closed circuit) happens when the positive terminal of diode is connected to a point which is at higher potential than the point to which negative terminal is connected. This condition is called forward biasing. On the other hand, the condition of minimum or zero current flow happens when positive terminal of diode is connected to a point which is at lower potential than the point to which the negative terminal of diode is connected.


A diode and its symbol

Before using any diode, you should always check if it is working fine by doing continuity test using a multimeter. The diode must offer continuous flow of current when its positive terminal is connected to red (positive) probe and its negative terminal is connected to black (negative) probe. The diode should offer discontinuity when multimeter is connected in opposite direction (that is Red probe to negative terminal of the diode and Black probe to positive terminal of the diode). Most commonly used diodes in electronics are 1N4001 and 1N4007.

## 2.2.6 LED

LEDs or light emitting diodes is a special category of diodes that emits light when forward biased. Generally, LEDs are of two types:

Visible light emitting LED ( simple LED)

Infrared light emitting LED (IR LED)
Visible light emitting LED

A typical LED has a current rating of 20 mA, life time of 2 lakh working hours, and can tolerate a maximum voltage of around 4 V. There are also coloured LEDs available in the market, and there colour is due the doping in the semiconducting material used to make them.

All LEDs have two terminals, one positive and the other negative. The negative terminal is of shorter length, but there are some LEDs with a longer negative terminal which may be due to manufacturing defect. Also, once you cut the lengths a LED to use it in a circuit, the identification by comparing lengths of terminals is not useful. Another way of identifying the positive and negative terminal is to compare the filament. Again, due to manufacturing defects, some LEDs have a broader positive terminal. So none of the ways mentioned can be relied upon completely. The best way is to use the multimeter. Set the multimeter to continuity checking mode and touch red probe to one terminal of LED and black probe to other terminal of LED. If the LED glows, the terminal in contact with the red probe is the positive terminal and other is the negative one. If the LED doesn't glow, change the polarity. Now, if the LED glows, the terminal now in contact with the red probe is positive and the other negative. However, if the LED still doesn't glow, it means the either the LED or the multimeter is faulty.

An LED, with the cathode on the left and anode on the right

LEDs are generally used in large number in circuit. They are used in parallel with a regular connection in a circuit, and this helps a lot during troubleshooting (that is finding the error in the circuit if it doesn't work). An LED glowing in parallel of a connection suggests that the current is flowing through the connection in the direction of the negative terminal from the positive terminal of LED. However, if the LED doesn't glow, it suggests that either the current is flowing in the opposite direction, or the current is not flowing at all in that part of the circuit. A thumb rule to be followed while using LEDs is to use a resistor of around 100 ohm in series with LED, so as to avoid the LED getting burnt in case the voltage across it exceeds 4V or current exceeds 20mA.

### Infrared light emitting LED (IR LED)

There are two differences between a normal LED and an IR LED. The first one, of course, is that an IR LED emits infrared radiations which cannot be seen with the naked eyes, whereas a LED emits visible light. Another difference is that IR LED consumes a lot of current and thus they get damaged very fast. They do not have lives as long as LEDs.

An important precaution to be taken while using an IR LED is to not allow it to be forward biased for a very long time as we know that infrared radiations carry a lot of heat. Continuous IR radiations may heat the body or shell of the IR LED too much and can damage it also. Be careful not to touch IR LED after it has been forward biased for a long time since it may have become very hot.

## 2.3 Multiple voltage requirements

Almost all the electronic chips and devices which we will be using would work at different voltages. Multiple voltages requirement can be handled in several ways; the easiest and most straightforward of them is to use a different set of batteries for each main subsection. However, it is cumbersome and impractical to provide a different power source having different voltage ratings to each of the component. This method is useful when the motors used in the robot draw a lot of current. Motors generally distribute a lot of electrical noise throughout the power lines. And sometimes, this is the noise to which the electronic circuitry is extremely sensitive. Thus if you use different set of batteries for motors and circuitry, the electrical isolation provided by it to the circuitry almost completely eliminates the problems caused by noise.

In addition, when the motors are started initially, the excessive current drawn from the motors may deprive the electronic circuitry of the necessary current, which may induce 'sag' that can cause erratic circuit behaviour, which may in turn cause your robot's processor to lose control. Despite these advantages, using different batteries for different components makes the robot too heavy.

The other approach to handling multiple voltages is to use one main battery source and multiple voltage regulators to step it up or down according to voltage requirements of individual components in the system. This method is called DC-DC conversion. You can accomplish DC-DC conversion by using your own circuits or by purchasing specialty integrated circuit chips that make the job easier. A 9V battery's output voltage can be changed to a wide range of voltages – from as low as tending towards zero to as high as 15 V. Normally, a single battery will directly drive the motors, and with proper conversion, supply 5V to the circuit boards. Connecting the batteries judiciously can also yield multiple voltage outputs. By connecting two 6V batteries in different ways, we can get 12V and 6V. This system isn't nearly as fool proof as it seems, however. More than likely, the two batteries will not be discharged at the same rate. This causes extra current to be drawn from one to the other, and the batteries may not last as long as they might otherwise. If all of the subsystems in your robot use the same batteries, be sure to add sufficient filtering capacitors across the positive and negative power rails. The capacitors help soak up current spikes and noise, which are most often contributed by motors. Place the capacitors as near to the batteries and the noise source as possible. A good rule of thumb is 100 µF of capacitance for every 250 mA of current drawn by a motor during normal operation. Be certain the capacitors you use are overrated above the voltage by 50 to 75 % (e.g. use a 7.5 V rated capacitor for a 5 V circuit). An underrated capacitor will probably burn out or possibly develop a short circuit, which might "fry" your circuit. You should place smaller value capacitors, such as 0.01 to 0.1 µF, across the positive and negative power rails wherever power enters or exits a circuit board. As a general rule, you should add one of these decoupling capacitors at the power input pins of all ICs. Linear ICs, such as the 555 timer need decoupling capacitors, or the noise they generate through the power lines can ripple through to other circuits.

### Voltage Regulation

Most circuits require a precise voltage or they may get damaged or act erratically. Generally, provide voltage regulation only to those components and circuit boards in your robot that require it. You can easily add a variety of different solidstate voltage regulators to your electronic circuits. They are easy to obtain, and you can choose from among several styles and output capacities. In this chapter, some of the different types will be described along with their operating characteristics.

### Zener diode voltage regulation

A quick and relatively small method for providing regulated voltage is to use zener diodes. With a zener diode, current does not begin to flow through the load circuitry until the voltage exceeds a certain level (called the breakdown voltage). Voltage over this level is then “shunted” through the zener diode, effectively limiting the voltage to the rest of the circuit. Zener diodes are available in a variety of voltages, such as 3.3, 5.1, 6.2 V and others. A zener diode and resistor can make a simple and inexpensive voltage regulator. Be sure to select the proper wattage for the zener and the proper wattage and resistance for the resistor. Zener diodes are also rated by their tolerance (1 percent and 5 percent are common) and their power rating, in watts. Note the resistor in the schematic shown in the figure given alongside.

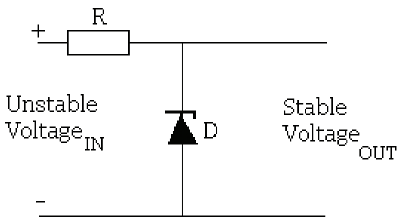
This resistor (R) limits the current through the zener, and its value (and wattage) is determined by the current draw from the load, as well as the input and output voltages. The process of determining the correct values and ratings for resistor R and the zener diode is fairly simple and uses the basic electricity rules. The zener voltage rating is, quite obviously, the desired regulated voltage—you may find that the available rated voltages are somewhat awkward (such as 5.1 V), but you should be able to find a value within a few percent of the rated value. Once you know the voltage rating for the zener diode that you are going to use, you can then calculate the value and ratings for R. The zener diode regulator shown in the figure alongside is actually a voltage divider, with the lower portion being a set voltage level. To determine the correct resistance of R, you have to know what the input voltage is and the current that is going to be drawn from the regulator. For example, if you wanted 100 mA at 5.1 V from a 12V power supply, the resistance of R can be calculated as:

VImax = 12-5.1100 ×10-3=69z

The closest "standard" resistor value you can get is 68 ohms, which will result in 101 mA being available for the load. With this value in hand, you can now calculate the power being dissipated by the resistor, using the basic power formulas:

V×Imax=6.9×100×10-3=0.69W

Standard resistor power ratings (in watts) are 18, 14, 12 , 1, 2, and so on. A 1-W, 68 ohm resistor would be chosen for this application. The zener diode will also be dissipating power. The voltage drop here is something that you need to decide. To be on the safe side, it is recommended that you assume that the zener diode can have 100 percent of the load current passing through it (when the load circuitry is not attached to the power supply). The power rating for the zener diode is calculated exactly the same way as R:



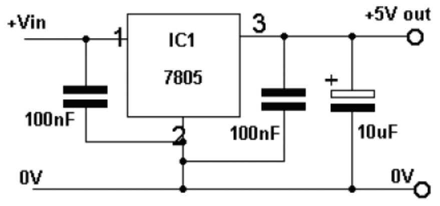
A simple Zener diode voltage regulator

5.1V×0.101A=0.52 W

For this application, you could probably get away with a ½ W rated zener diode along with the 1 W rated 68 ohm resistor. Working with zener diodes to make power supplies is quite easy, but there is a tremendous price to pay in terms of lost power. In this example, the total power dissipated will be 1.2 W, with 58 percent of it being dissipated through the R resistor. This loss may be unacceptable in a battery-powered robot.

## Linear voltage regulators

The zener diode regulator can be thought of as a tub of water with a hole at the bottom; the maximum pressure of the water squirting out is dependent on the level of water in the tub. Ideally, there should be more water coming into the tub than will be ever drawn to ensure that the pressure of the water coming out of the hole is constant. This means that a fair amount of water will spill over the edge of the tub. As was shown in the previous section, this is an extremely inefficient method of providing a regulated voltage, as the electrical equivalent of the water pouring over the edge is the power dissipated by R.



To improve upon the zener diode regulator's inefficiency, a voltage regulator that just lets out enough current at the regulated voltage is desired. The linear voltage regulator only allows the required current (at the desired voltage) out and works just like a car's carburettor. In a carburettor, fuel is allowed to flow as required by the engine—if less is required than is available; a valve closes and reduces the amount of fuel that is passed to the engine. The linear voltage regulator works in an identical fashion: an output transistor in the regulator circuit only allows the required amount of current to the load circuit. The actual circuitry that implements the linear regulator is quite complex, but this is really not a concern of yours as it is usually contained within a single chip like the one shown in figure below.

The circuit shown here uses one of the most popular linear voltage regulators, the 7805 to regulate a high-voltage source to 5 V for digital electronic circuitry. Two of the most popular voltage regulators, the 7805 and

7812, provide 5 and 12 V, respectively. Other 7800 series power regulators are designed for 15, 18, 20, and 24 V. The 7900 series provides negative power supply voltages in similar increments. The current capacity of the 7800 and 7900 series that come in the TO-220 style transistor packages (these can often be identified as they have no suffix or use a “T” suffix in their part number) is limited to less than 1 A. As a result, you must use them in circuits that do not draw in an excess of this amount.

## 2.4 The Microcontroller

A microcontroller (or a microprocessor) is a device which receives the signal given by the sensors and processes it. A microcontroller has a predefined set of conditions and corresponding actions to be taken for each of them.

| If voltage given by sensor is (conditions) | Do the following (Instructions) |
|---|---|
| Less than 3 volts | Give 5 volt at the output |
| Equal to 3 volts | Give 6 volt at the output |
| Greater than 3 volts | Give 7 volt at the output |

Processing a signal essentially means checking that with which condition does the input matches. For example, if the input given by the sensors to the microcontroller is 2 V, the microcontroller will immediately provide 5 volt at its output. Many vendors manufacture microcontrollers. The most popular products and their vendors are given in the following table:

| Name | Manufacturer | Comments |
|---|---|---|
| PIC | Microchip | Huge product line that is very popular with the robot makers. The PIC16F84 has been a one of the favourites of hobbyists |
| AVR | Atmel | Becoming increasingly popular part with robot enthusiasts. Good free tools available for standard and MegaAVR parts. |
| H8 | Hitachi | Mainly used in many commercial robots. Tools available on the Internet but fewer example applications than available for other devices. Not quite famous for robotic projects. |
| 8051 | Intel (originally) | One of the oldest, and thus many different varieties and part numbers available from a lot of manufacturers. The original parts are very simple but now different manufacturers have MCUs with sophisticated interfaces. Lots of examples, tools and support available on the Internet. |

All of the above mentioned vendors provide excellent support in terms of software and all the above mentioned devices can be used with ease once you get acquainted with them.

## 2.5 Op-amp

Op-amp stands for operational amplifier. It is just like an amplifier which you use while listening to music. It performs the function of amplifying the voltage signal input to it, converting it into a voltage of significantly higher magnitude. Generally, it comes in form of an IC and it has multiple ports to take multiple signals and amplify them simultaneously. The most popular Op-amps among hobbyists are MCP series and LM series Op-amps. We are going to use LM358N Op-amp for our purpose.

## 2.6 Motor driver

A motor driver is an IC which is capable of providing voltages sufficient enough to drive D.C. motors. We use a L293D motor driver. The L293D is designed to provide bidirectional drive currents of up to

600mA at voltages from 4.5 V to 36 V. It can also drive inductive loads such as relays, solenoids, dc and bipolar stepping motors, as well as other high-current/high-voltage loads in positive-supply applications.

## 2.7 Sensors

Robots are very similar to human body. Human body has senses to receive information from the external environment. Robots have sensors. Human body has brain to process the information; robots have processors to do the same. Human body have the external organs to execute decisions given by mind, in robots the wheels or legs or arms does it. There are two broad categories of sensors:

Digital sensors: They provide discrete or stepped binary results. A switch an example of a digital sensor: It can either be open or closed. Also, an ultrasonic ranger, which returns a different binary value for distances, is also a type of digital sensor.

Analogue sensors: They provide a continuous range of values, usually in form of voltage.

Generally, in both digital and analogue sensors, the input to the processing unit is voltage. In the case of a digital sensor, the processing unit is interested in knowing whether the voltage input given by the sensor is a logic low (usually 0 V) or logic high (usually 5 V). Most of the digital sensors can be directly connected to the processing unit without any additional interfacing electronics, but in case of an analogue sensor, you need additional circuitry to convert the varying voltage levels into a form that the processing unit can use. Also, circuitry is required to amplify the output of the analogue sensor, since this output may not be adequate to be interpreted by the processor. This is basically achieved by using operational amplifiers or op-amps as we have discussed.

**Types of Sensors**

In robotics projects, the following types of sensors are widely used:

Bump or touch sensors: These are those sensors which are activated when there is touch or bump.

Temperature sensors: These are just like thermometer, and they respond to change in temperature by changing voltage across their terminals. Commonly available temperature sensors are LM35, DS1621 and a thermistor.

Light sensors: These are those sensors which are activated whenever there is significant change in the intensity of light. The most commonly used light sensors are the IR (Infrared) sensors and LDR (Light Dependent Resistor) sensors.

**IR Sensors**

Appliances such as TVs, music players and ACs have IR transmitter-receiver pairs to transmit signals.

Colour sensors: These are advanced light sensors that can sense even the slightest change in the intensity of light falling on them and thus can detect colours. This is because each colour reflects a different intensity of light when light falls on them. So when there is a change in the intensity of light falling and light reflected, the sensor senses it and sends the signal.

Sonar proximity detector: In this, the reflected sound waves are used to judge distances between the robot and obstacle, with the aid of sound signals.

Speech input or recognition: Various sound/speech patterns or even a human voice can be used to command the robot.

Gas or smoke sensor: These detect noxious or toxic fumes once their concentration in air increases beyond a certain limit.

Pyroelectric infrared sensor: It detects changes in heat patterns over period of time and is thus used as motion detectors.

Sound sensor: Robot responds to sound patterns like a clap or drum beat. The robot can also be tuned to listen to only certain sound frequency (which may also lie outside a human's audible range) or to those sounds having a certain minimum volume level.

Accelerometer: These are used to detect changes in inclination or speed of the robot.

Most of these sensors are available in the ready to use form in the market, but they are quite expensive because they offer very good precision. But for robotics enthusiasts, buying them is not always a good deal because you can construct majority of them on your own and those too offering reasonably good precision and efficiency.

## 2.8 Input and output methodologies

The changes sensed by the sensors are converted into voltage difference and this voltage difference is then sent to the "mind" of the robot, which is a microcontroller (or microprocessor). The robot's processing unit requires input electrical connections to receive the signals from sensors, as well as connections to outputs like a motor. These signals are in form of bits (BInary digiTS). Two types of interfaces are used to transfer these signals (or bits) from sensors to the robot's control computer or from robot's control computer to motors:

### Parallel Interfacing

In a parallel interface, multiple bits of data are transferred simultaneously by eight separate wires. This type of data transfer offers high speed because more information can be transferred in less time. A typical parallel interface commonly used in day to day life is a printer port on a personal computer; it sends the data to the printer byte by byte (1byte=8 bits). Different characters including alphabets, numbers and symbols are represented by different combinations of the eight-bit data (as according to ASCII character codes).

### Serial Interfacing

Though parallel interfacing offers more speed, they occupy a large number of input/output ports. This proves to be a serious disadvantage when you have limited number of input/output ports and can't expand their number.

Serial interfaces, on the other hand, send data on a single wire, and thus conserve on number of input/output ports used. They do this by sending each bit in a queue (one by one). Obviously, this process is much slower than the parallel data transmission. There is a variety of serial interface schemes available, using one, two, three, or even four input/output lines.

These additional input/output lines are used for tasks such as timing and coordinating between the data sender and the data recipient, so that the data does not get lost or abstruse while being transferred.

For robotics application, the data transfer rate need not be too high, and on a microcontroller, there is always scarcity of input/output ports. So, serial transmission is generally preferred in basic robotics applications. Although implementation of serial interface may seem difficult, it's not so if you use the proper software and hardware. The software works with the incoming serial data and the whole task of recombining the bit wise information into a byte, is performed by the processor. d

# 3 Introduction to Mechanics

In this chapter, we cover the mechanical aspects of making robot, including selection of the robot building material from the choices available, mounting of motors and other peripherals on the body.

## 3.1 Centre of mass

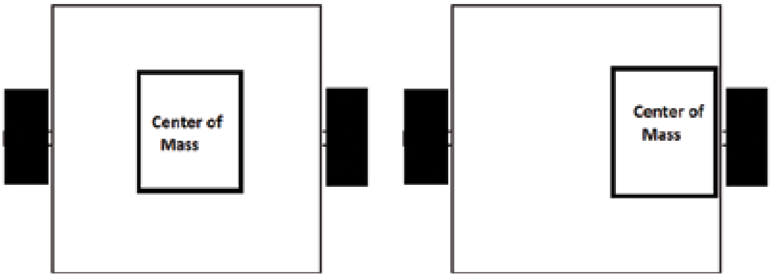
Centre of mass of a body is a point at which you can assume the whole mass of the body to be situated for analytical purposes. There are two centre of masses defined for a body: horizontal centre of mass and vertical centre of mass.

### Horizontal centre of mass

As we all know, symmetry increases stability. If most of the weight of your robot is not located at the centre but concentrated to one side, the robot will be unstable. This may occur when you place your batteries, which are generally the heaviest components of your robot on one side and not at the centre. For most stable configurations, the horizontal centre of mass should coincide with the centre of the robot's body. If there is any shift in the centre of mass from the centre of body, the robot may tip over or topple.

Also, if the weight is concentrated mostly over one wheel, the grip of that wheel will be more than that of the wheel on which no significant weight is kept, because of which the robot may not be able to move straight.

A robot with a small base and high vertical center of gravity can topple easily. So, to bring necessary stability, you can design your robot in two ways: Having low height and small area of base, or, having good height, but also



Vertical centre of gravity (or mass)

large area of base. The choice between the aforementioned methods depends on what you want your robot to do. For a robot, which must keep and pick up things from table, you need to have good height and thus method two is apt, while for a line follower where big height is of no use, the first method is the best.

Before we go on to design the body of the robot, we need to learn some basic aspects of locomotion.

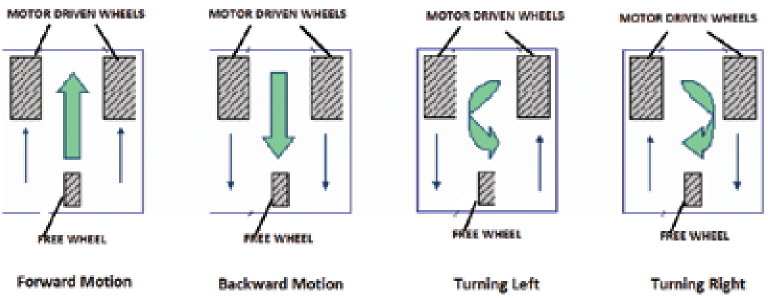
## 3.2 Types of wheeled locomotion systems

### 3.2.1 Differential drive

This is perhaps the most commonly used type of locomotion systems in mobile robots, because it is simple and easy to design as well as implement. This type of structure balances itself on three wheels; two of them are motor driven, while the third one is free. The free wheel, also called pivot wheel, can be mounted at the front or at the back of the pair of wheels driven by the motors. Practically, the pivot wheel is best mounted at the back, so it acts as a 'trailing' wheel and not as a 'steering' wheel.

When both the motor driven wheels rotate in the same direction, the robot moves straight. If both wheels rotate in clockwise direction, and the robot moves forward, then when both wheels rotate in anticlockwise direction, the robot will move backward.

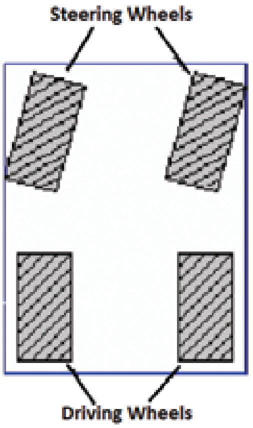
The robot turns when the motor driven wheels move in opposite direction. When left wheel moves clockwise and right wheel moves anticlockwise (as seen from the right side of robot), the robot turns right. On the other hand, when right wheel moves clockwise and left moves anticlockwise (again, as seen from right side of the robot), the robot turns left.



In differential type drive, zero radius of turning can be achieved by rotating the wheels at same rate in opposite direction. Another point to be noted here is that this design requires the use of two independent motors, and both of which are used in both rotational and translational motion.

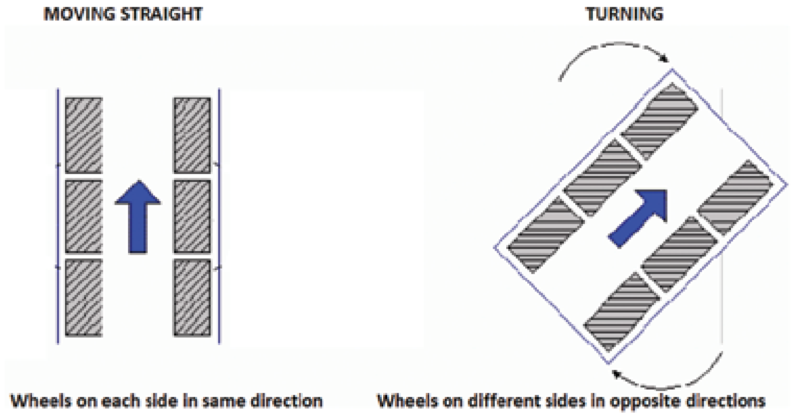
### 3.2.2 Car type design

This design makes use of four wheels, two of them at the front for steering, and the rest two at the back to provide forward/ backward motion. This design type is very common in real world, and almost all the four wheeled vehicles make use of this design. However, this design type is fairly complex and thus is generally avoided in basic robotics.



### 3.2.3 Skid steer drive

This type is very similar to the differential drive design type. There are multiple numbers of driving wheels present in this, and there is no need of pivot wheel. The left and right groups of wheels are driven independently. This type of drive is commonly utilized in military tanks. In robotics, we make use of this type of design when there is a need of high friction and grip on the surface, for example, they are widely used in wall climbing robots. Turning


MOVING STRAIGHT
TURNING
Wheels on each side in same direction
Wheels on different sides in opposite directions

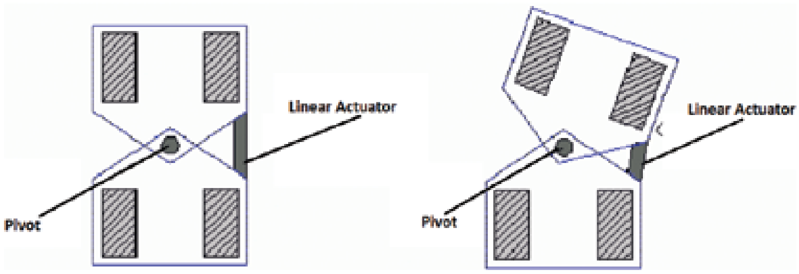
in this type of design is achieved by moving the set of wheels on each side at different rate or in different directions.

### 3.2.4 Articulated drive

Probably it is the most complex design which can be used for locomotion. The body of the robot is modified to achieve rotation. Two motors are needed in this design, one of them used for translation, the other used to change the pivot direction. Since the body of the robot is moving internally too, mounting the circuit board and batteries on this type of structure is difficult, and hence it requires lot of planning. This type of design is followed in 'snake type robots'.

### 3.2.5 Synchronous drive

This type of design relies on synchronous movement of all the wheels to achieve motion as well as turning.  Generally, this type of design is consists of 2 sets of motors, one set used to drive the wheels, the other set used to turn the wheels simultaneously at same angle.

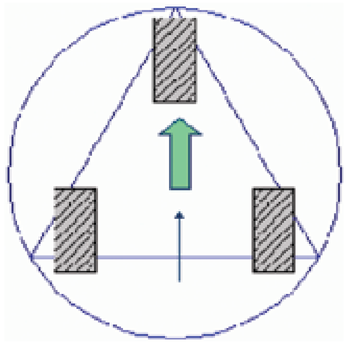


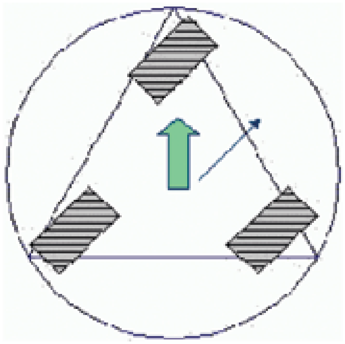
## 3.3 Material for body

To build a robot's body, we need to select a building material for it. Following are some of the choices available:

### 3.3.1 Paper

Paper and cardboard are often overlooked as far as choosing the material to make a robot's body is concerned, but they can prove to be handy as they are easy to work with and give faster results. There are lot of varieties of paper and paper products which can serve as robot body building materials. Moreover, you would find them in your home itself and you won't have to





spend money on them at all.

There are a number of different papers, cardboards and foam-backed boards that could be used as part of a robot's structure; all you require is a bit of imagination. You may not have thought of this, but paper and cardboard which serve as packing materials for appliances are really strong and tough and thus are excellent materials for building your robots and different parts of them. Also, you won't require any sophisticated and complex tools like a drill or a lathe or welding machine, you can just do everything with a pair of scissors, a cutter, some glue and adhesive tape. The parts made from a piece of cardboard are much easier to ensure proper fit and clearances with other parts. Many people first make a prototype of the desired robot with cardboard and after they are done with it, they make the final robot with materials like wood or metals.

The only disadvantage to use cardboard as material for your robot is its ability to soak water and moisture, and this proves very critical if you are designing a robot which will traverse through wet or moist surfaces.

### 3.3.2 Wood

One of the most easily available materials for making robots, wood has always been the favourite for robot builders .There are many different wood varieties and products to choose from for use in a robot's structure and each one of them has its own characteristic feature that affect its suitability in different parts of the robot's body. The biggest advantages of using a wood are its ability to be worked using inexpensive hand tools like wood saw and also its fairly low cost. If you plan to use wood as a material to build the structure of your robot, you are advised to use the hardest woods available (such as oak). Very hard woods will handle high loads comfortably and, more importantly, will not split at bolt attachment points. Soft woods like balsa should always be avoided. Also, while fastening the pieces of wood to other pieces, nuts and bolts should be used instead of screws to minimize the chance that the wood will be damaged or the holes will open up due to vibration.

Along with solid pieces of wood, you may also consider using a composite wood product like plywood, which consists of many sheets of wood glued together under pressure in such a way that its strength is augmented. Plywood is often manufactured from softer woods like spruce and the final product is much stronger than the sum of its constituents.

However you should not use composite materials like chipboard, as they cannot sustain much vibration and thus are inappropriate for robot body building purposes.

### 3.3.3 Plastics

Probably the number of plastic varieties available in the market is far greater than that of all varieties of other materials combined. Determining the correct type of plastic for a specific application can be a very arduous task and may take an unreasonably long period of time. Instead, if you want to work with plastics, it's better to pick a random variety which is available to you easily and start working on it, because all varieties of plastics are generally strong and tough, and thus are well suited for robot building applications.

Still, there are two most popular types of plastics you can consider:

#### Thermoset plastics

They are hard, tough and have a compactly meshed molecular structure. They are generally hardened in a mould using chemicals, and during this process they tend to release a lot of heat. This type of plastics can only be shaped by machining as heating them up will destroy the molecular structure and will make the material useless. Thermosets are generally used in hard plastic toys, appliances, furniture and other products which require high structural strength.

#### Elastomer plastics

They have a similar molecular structure to thermosets, but they have elastic characteristics also. Elastomers too, like thermosets, cannot be shaped up by heating.

### 3.3.4 Metals

Metals are used for building structures of robots mainly in industry, but some enthusiasts do use them for making their robots because of the sheer strength they provide. You may get any metal you want at junk shop, but it's always better to avoid choosing metal as robot building material if you are a beginner and do not have the luxury of a workshop at your disposal. Most commonly used metal for robot building is aluminium which is light, offers good strength, and is also immune to corrosion.

### 3.3.5 Motors

A motor is a mechanical device which converts electrical energy into mechanical energy. Any robot, which needs to move, must have at least one motor. If you want your robot to move like a car, attach motor with wheels to your robot. If you want your robot to have a mechanical arm or wrist, put a motor where you want a joint. Anywhere you need motion, motor is there to help.

There are many kinds of motors available, but only some of them are utilised for robotics.

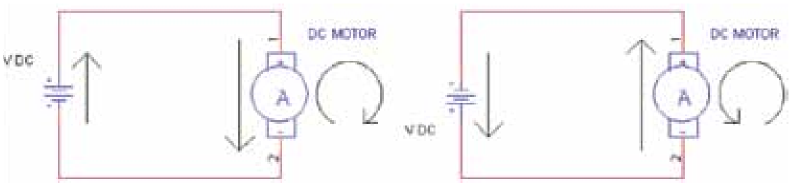
This section will examine the various kinds of motors, how they are used and where they can be used.

There are two broad categories of motor, based on the form of power supply they require: AC motors (works on alternate current ) and DC motors (works on direct current).

Mainly, direct current dominates the field of robotics, especially for amateurs and hobbyists. This is because, in robotics, we prefer on-board power source, which can be provided by DC batteries very easily. Mainly industrial robots use motors designed to operate from AC. Some of those robots convert the AC power to DC, and then utilize it, while some directly use AC current, and thus require the use of AC motors.

There are many kinds of DC motors available in the market, and you should buy a DC motor which suits your application the best. For example, you should always buy reversible motors (those which can rotate in both directions) because very few robotic applications call for just unidirectional (one-direction) motors. Generally DC motors are bidirectional, but some design limitations may restrict reversibility. The most prominent factor is the commutator brushes. If the brushes are slanted, the motor probably can't be reversed. Another factor which may prevent the reversibility is the internal wiring of some DC motors. These factors may be present in the form of manufacturing defects, and spotting them by just looking at the motor is very difficult. The best way to check it is to power the motor with a suitable battery. Apply the power leads from the motor to the terminals of the battery or supply and note the direction of the rotation of the shaft. When you reverse the power leads from the motor, the motor shaft should rotate in reverse direction.

DC motors can be either continuous or stepper. In a continuous DC motor, the application of continuous power causes the shaft to rotate continually, while in a stepper the application of power causes the shaft to rotate a few



degrees, then pause, and then rotate again. In the continuous motor, the shaft stops only when the power is removed or if the motor is stalled, which means that the load is too much for the motor and it cannot drive it. Each 'step' of rotation of a stepper motor is precisely of a fixed angle of rotation.

### Motor specifications

There are four parameters, which you should consider while buying a motor for yourself - voltage, current draw, speed, and torque.

### Operating (or rated) voltage

Operating voltage gives the value of voltage which when applied across its terminals, produces best efficiency with a small DC motor, the rating is usually a range, like 1.5 to 6 V. Some high-quality DC motors are designed for a specific voltage, such as 12 or 18 V. But the kinds of motors of most interest to robot builders are the low-voltage variety—those that operate at 1.5 to 12 V.

Most motors can be operated satisfactorily at voltages higher or lower than those specified. A 12 V motor is likely to run at 8 V, but it may not be as powerful as it could be and it will run slower. If you use very less voltage (around under 50 percent of the specified rating) as compared to its rated voltage, the motor will stop working. Also, a 12V motor is likely to be able to run at 16 V. As the voltage across its terminal increases, speed of the shaft rotation increases. But if you want to increase the speed, you just cannot keep on increasing the voltage across its terminals, as if you go beyond the rated voltage, the internal coil will burn and the motor will be damaged. So, it is always better to operate around the rated voltage.

#### Current draw

Current draw is the amount of current that the motor draws from the power supply. For small DC motors, it is measured in in milliamps. The current drawn by a motor when no load is applied to it can be quite low. But when the load is applied, the current draw becomes three or 4 times, depending upon the load applied. Most of the DC motors you will encounter in the market are permanent magnet motors, and in all of them current drawn increases with the load applied. If you keep on increasing the applied load, a stage is reached when the motor shaft doesn't rotate at all. No more current flows through it. This condition is called motor stalling. Some motors are rated by the manufacturer by the amount of current they would draw when stalled.

#### Speed

The rotational speed of a motor is given in revolutions per minute (rpm). You can get DC motors having a normal operating speed of 10 rpm to those having operating speed of 7000 rpm. For robotic applications, you would need very low rpm motors, and it's unlikely that you would ever go beyond a 500 rpm motor in robotics.

#### Torque

Torque is the force applied by a revolving axle. The higher the torque, the larger is the force applied. You can actually test the torque of a motor by trying to rotate its axle when it is not powered by batteries. The more power you require, the more torque that particular motor has. If the torque of the motor is less compared to the load being applied, the motor would simply stall and consume a lot of current and dissipating a lot of heat.

### 3.3.6 Battery

If you want your robot be to be completely self-operated, and you do not want any wire coming out of the robot even for getting power, you need to provide on board batteries. Many different types of batteries are available in the market, NiCad or NiMH batteries being most common.

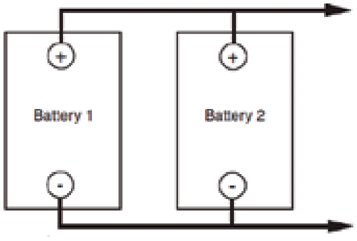
You can obtain higher voltages and current by connecting several cells together, and there are two different approaches to do that:

To increase voltage, connect the batteries in series. The resultant voltage is the sum of the voltage outputs of all the cells combined.

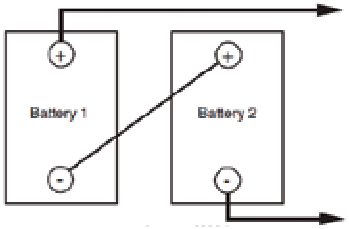
To increase current, connect the batteries in parallel. The resultant current is the sum of the current capacities of all the cells combined.

A point to be noted here that when you connect cells together not all cells may be discharged or recharged at the same rate, like the case when you have some new batteries connected with some half used batteries. The new batteries would provide most of the power and would get discharged earlier than the half used batteries.

Therefore, you should always replace or recharge all the cells together.



Parallel Connection (Current becomes twice)



Series connection (Voltage becomes twice)

Also, you can use pencil cells in series to get 9V or 12V (whatever voltage you require). Generally, batteries form the heaviest part of the robot, and so it is always advisable to keep the batteries as low as possible. Doing this would lower down the centre of mass of your robot and will make it more stable and less prone to toppling.

Taking care of your batteries and maintaining them for long time is very important. Follow the instructions given below to maintain your batteries strength for a long time:

Always keep your batteries dry. Do not short the battery terminals.

Remove battery from the holder when they are not in use. In case of NiCd batteries, discharge them fully before charging again, this would avoid development of memory effect in them. Do not expose batteries to high temperatures if you use them in a robot, keep the electronics devices which get heated very fast (like voltage regulators) away from them. Fix your battery to the base of your robot with Velcro, since Velcro has a good hold to keep batteries in place, and you can also remove the batteries easily if you want to replace or recharge them.

### 3.3.7 Wheels

Wheels when attached to the axle of motor provide locomotion. There are certain aspects of wheel you should consider before buying them:

#### Wheel texture

The texture of the wheel decides how much grip your wheel would have on the surface it moves upon. If the surface is very smooth, then you need to have wheels which have rubber coating on them, so that they can get good grip. If the surface itself is too rough, you can afford to use plastic wheels. However, it's always better to use rubber coated wheels as then you don't need to worry much about the surface.

#### Wheel diameter

Wheels with large diameter give you lower torque but higher velocity. If you have a motor with high torque, you can make use of high diameter wheels. Also, high diameter wheels are required when you need large ground clearance, which is defined as the height of the base of your robot from the ground. So if you plan to make an 'All Terrain Type' of robot, use rubber coated wheels with big wheels. However, if you don't need too much ground clearance you can use small wheels in your robot. A point to keep in mind is that the diameter of the wheel must always be bigger (by at least 3-4 cm) than the diameter of the motor, and if they are equal or nearly equal, the ground clearance would be zero and the base of the robot would touch the ground.

#### Wheel center hole diameter

The center hole is the slot where the shaft of the motor would be inserted. So, the diameter of this slot in your wheel must be greater than the diameter of the motor shaft. You can still manage if the wheel center hole is too big for a shaft, as then you can put in some paper or glue to jam them together, but if the slot in the wheel is smaller than the motor shaft, the shaft would simply not enter it.

# 4 Introduction to programming

A program or a code is a predefined set of instructions which are fed into the memory of a processing unit. These instructions define what the processing unit would do under different input conditions. This set of instructions is called a 'code' or a 'program'.

If you have to instruct someone to do some task, you would need to have a medium through which you can convey your idea. The medium can be a language or a sign or something else, but it must be understood by you as well as the person whom you want to instruct. Similarly, to instruct a microcontroller you need a common language or a platform, where you can exchange instructions or information.

Generally, all the machines understand 'binary language', which is composed of only 0's and 1's. Unfortunately, it is very difficult for us to convey instructions in binary form for us. And machines also do not understand any of the languages we speak.

A common language, thus, is required. These common languages are called 'programming languages' and the most commonly used are C, C++ and JAVA. Here we would use C++.

Given below is a sample C code for sampling data over RS-232 interface.

```
  ///////////////////////////////////////////////////////
  ///  This program displays the min and max of 30,   ///
  ///  comments that explains what the program does,  ///
  ///  and A/D samples over the RS-232 interface.     ///
  ///////////////////////////////////////////////////////

  #if defined(__PCM__) // preprocessor directive that
chooses the compiler
  #include <16F877.h> // preprocessor directive that
selects the chip PIC16F877
  #fuses HS,NOWDT,NOPROTECT,NOLVP // preprocessor directive
that defines fuses for the chip
  #use delay(clock=20000000) // preprocessor directive
that specifies the clock speed
  #use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7) //
preprocessor directive that includes the rs232 libraries
```

```
  #elif defined(__PCH__) // same as above but for the PCH
compiler and PIC18F452
  #include <18F452.h>
  #fuses HS,NOWDT,NOPROTECT,NOLVP
  #use delay(clock=20000000)
  #use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)
  #endif
  void main() {  // main function
  inti, value, min, max;  // local variable declaration
  printf("Sampling:"); // printf function included in the
RS232 library
  setup_port_a( ALL_ANALOG ); // A/D functions- built-in
  setup_adc(  ADC_CLOCK_INTERNAL  );  //  A/D  functions-
built-in
  set_adc_channel( 0 );   // A/D setup functions- built-in
  do {                     // do while statement
  min=255;              // expression
  max=0;
  for(i=0; i<=30; ++i) {    // for statement
  delay_ms(100);           // delay built-in function call
  value = Read_ADC();      // A/D read functions- built-in
  if(value<min)             // if statement
  min=value;
  if(value>max)             // if statement
  max=value;
  }
  printf("\n\rMin: %2X  Max: %2X\n\r",min,max);
  } while (TRUE);
  }
```

This code can be only compiled by CCS compilers since it uses CCS specific functions and pre-processor directives. Though it may look daunting, after you finish with this section, you'll be able to understand it.

## 4.1 Concepts

This part covers some basic concepts of the language, essential for understanding how C programs work. First, we need to establish the following terms that will be used throughout the help:

### 4.1.2 Objects

An object is a specific region of memory that can hold a fixed or variable value (or set of values). To prevent confusion, this use of the word object is different from the more general term used in object-oriented languages. Our definition of the word would encompass functions, variables, symbolic constants, user-defined data types, and labels.

Each value has an associated name and type (also known as a data type). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely references the object.

#### lvalues

An lvalue is an object locator: an expression that designates an object. An example of an lvalue expression is *P, where P is any expression evaluating to a non-null pointer. A modifiable lvalue is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A constant  pointer to a constant, for example, is not a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

#### Rvalues

The expression a + b is not an lvalue: a + b = a is illegal because the expression on the left is not related to an object. Such expressions are sometimes called rvalues (short for right values).

### 4.1.3 Scope and visibility

#### Scope

The scope of identifier is the part of the program in which the identifier can be used to access its object. There are different categories of scope: block (or local), function, function prototype, and file. These depend on how and where identifiers are declared.

Block: The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the enclosing block). Parameter declarations with a function definition also have block scope, limited to the scope of the function body.

File: File scope identifiers, also known as globals, are declared outside of all blocks; their scope is from the point of declaration to the end of the source file.

Function: The only identifiers having function scope are statement labels.

Label names can be used with goto statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing label_name: followed by a statement. Label names must be unique within a function.

Function prototype: Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.

### Visibility

The visibility of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object. Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Technically, visibility cannot exceed scope, but scope can exceed visibility. Take a look at the following example:

```
void f (inti) {
int j;         // auto by default
 j = 3;        // inti and j are in scope and visible

   {                // nested block
double j;     // j is local name in the nested block
    j = 0.1;      // i and double j are visible;
                    // int j = 3 in scope but hidden
   }
                    // double j out of scope
 j += 1;           // int j visible and = 4
}
// i and j are both out of scope
```

### 4.1.4 Name spaces

Name space is the scope within which an identifier must be unique. C uses four distinct categories of identifiers:

1. goto label names. These must be unique within the function in which they are declared.

2. Structure, union, and enumeration tags. These must be unique within

the block in which they are defined. Tags declared outside of any function must be unique.

3.Structure and union member names. These must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.

4.Variables, typedefs, functions, and enumeration members. These must be unique within the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

Duplicate names are legal for different name spaces regardless of scope rules.

For example:

```
int blue = 73;
{ // open a block
enumcolors { black, red, green, blue, violet, white } c;
    /* enumerator blue = 3 now hides outer declaration
of int blue */
structcolors { int i, j; }; // ILLEGAL: colors duplicate
tag
    double red = 2;     // ILLEGAL: redefinition of red
}
blue = 37;                          // back in int blue scope
```

### 4.1.5 Functions

Functions are central to C programming. Functions are usually defined as subprograms which return a value based on a number of input parameters. Return value of a function can be used in expressions – technically, function call is considered to be an expression like any other. C allows a function to create results other than its return value, referred to as side effects. Often, function return value is not used at all, depending on the side effects. These functions are equivalent to procedures of other programming languages, such as Pascal. C does not distinguish between procedure and function – functions play both roles.

Each program must have a single external function named main marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions have external linkage by default and are normally accessible from any file in the program. This can be restricted by using the static storage class specifier in function declaration.

### Function declaration

Functions are declared in your source files or made available by linking precompiled libraries. Declaration syntax of a function is:

```
typefunction_name(parameter-declarator-list);
```

The function_name must be a valid identifier. This name is used to call the function.

The type represents the type of function result, and can be any standard or user-defined type. For functions that do not return value, you should use void type. The type can be omitted in global function declarations, and function will assume int type by default.

Function type can also be a pointer. For example, float* means that the function result is a pointer to float. Generic pointer, void* is also allowed.

Function cannot return an array or another function.

Within parentheses, parameter-declarator-list is a list of formal arguments that function takes. These declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. If the list is empty, function does not take any arguments. Also, if the list is void, function also does not take any arguments; note that this is the only case when void can be used as an argument's type. Unlike with variable declaration, each argument in the list needs its own type specifier and a possible qualifier constant or volatile.

### Function prototypes

A given function can be defined only once in a program, but can be declared several times, provided the declarations are compatible. If you write a nondefining declaration of a function, i.e. without the function body, you do not have to specify the formal arguments.

This kind of declaration, commonly known as the function prototype, allows better control over argument number and type checking, and type conversions. Name of the parameter in function prototype has its scope limited to the prototype. This allows different parameter names in different declarations of the same function:

```
   /* Here are two prototypes of the same function: */
   int test(const char*)    /* declares function test */
   int test(const char*p)   /* declares the same function
test */
```

Function prototypes greatly aid in documenting code. For example, the function Cf_Init takes two parameters: Control Port and Data Port. The question is, which is which? The function prototype `voidCf_Init(char *ctrlport, char *dataport);` makes it clear. If a header file contains function prototypes, you can that file to get the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is used only for any later error messages involving that parameter; it has no other effect.

**Function definition**

Function definition consists of its declaration and a function body. The function body is technically a block – a sequence of local definitions and statements enclosed within braces {}. All variables declared within function body are local to the function, i.e. they have function scope.

The function itself can be defined only within the file scope. This means that function declarations cannot be nested.

To return the function result, use the return statement. Statement return in functions of void type cannot have a parameter – in fact, you can omit the return statement altogether if it is the last statement in the function body.

Here is a sample function definition:

```
/* function max returns greater of its 2 arguments: */
int max(int x, int y) {
  return (x>=y) ? x : y;
}
```

`Here is a sample function which depends on side effects rather than return value:`

```
/* function converts Descartes coordinates (x,y) to
polar (r,fi): */
#include <math.h>
void polar(double x, double y, double *r, double *fi) {
  *r = sqrt(x * x + y * y);
  *fi = (x == 0 && y == 0) ? 0 : atan2(y, x);
  return; /* this line can be omitted */
}
```

**Functions reentrancy**

Limited reentrancy for functions is allowed. The functions that don't have their own function frame (no arguments and local variables) can be called both from the interrupt and the "main" thread.

Functions that have input arguments and/or local variables can be called only from one of the before mentioned program threads.

## 4.2 Types

C is strictly typed language, which means that every object, function, and expression needs to have a strictly defined type, known in the time of compilation. Note that C works exclusively with numeric types.

The type serves:

to determine the correct memory allocation required initially.

to interpret the bit patterns found in the object during subsequent access.

in many type-checking situations, to ensure that illegal assignments are trapped.

CCS supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating-point numbers in various precisions, arrays, structures, and unions. In addition, pointers to most of these objects can be established and manipulated in memory.

The type determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object's storage allocation. A given data type can be viewed as a set of values (often implementation-dependent) that identifiers of that type can assume, together with a set of operations allowed on those values. The compile-time operator, sizeof, lets you determine the size in bytes of any standard or user-defined type.

The CCS standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that CCS can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

### 4.2.1 Type categories

Common way to categorize types is to divide them into:

fundamental

derived

The fundamental types represent types that cannot be separated into smaller parts. They are sometimes referred to as unstructured types. The fundamental types are void, char, int, float, and double, together with short, long, signed, and unsigned variants of some of these.

The derived types are also known as structured types. The derived types include pointers to other types, arrays of other types, function types, structures, and unions.

### Fundamental Types
The fundamental types represent types that cannot be divided into more basic elements, and are the model for representing elementary data on machine level. The fundamental types are sometimes referred to as unstructured types, and are used as elements in creating more complex derived or user-defined types.

### Arithmetic Types
The arithmetic type specifiers are built from the following keywords: void, char, int, float, and double, together with prefixes short, long, signed, and unsigned. From these keywords you can build the integral and floating-point types.

### Integral Types
Types char and int, together with their variants, are considered integral data types. Variants are created by using one of the prefix modifiers short, long, signed, and unsigned.

The table below is the overview of the integral types – keywords in parentheses can be (and often are) omitted.

The modifiers signed and unsigned can be applied to both char and int. In the absence of unsigned prefix, signed is automatically assumed for integral types. The only exception is the char, which is unsigned by default. The keywords signed and unsigned, when used on their own; mean signed int and unsigned int, respectively.

The modifiers short and long can be applied only to the int. The keywords short and long used on their own mean short int and long int, respectively.

| Type | Size in bytes | Range |
|---|---|---|
| (unsigned) char | 1 | 0 to 255 |
| signed char | 1 | - 128 to 127 |
| (signed) short (int) | 1 | - 128 to 127 |
| unsigned short (int) | 1 | 0 to 255 |
| (signed) int | 2 | -32768 to 32767 |

| | | |
|---|---|---|
| unsigned (int) | 2 | 0 to 65535 |
| (signed) long (int) | 4 | -2147483648 to 2147483647 |
| unsigned long (int) | 4 | 0 to 4294967295 |

**Floating-point types**

Types float and double, together with the long double variant, are considered floating-point types. CCS's implementation of ANSI Standard considers all three to be the same type.

Floating point in CCS is implemented using the Microchip AN575 32-bit format (IEEE 754 compliant).

The table below is the overview of the floating-point types:

| Type | Size in bytes | Range |
|---|---|---|
| float | 4 | ✥1.17549435082 * 10-38 to ✥6.80564774407 * 1038 |
| double | 4 | ✥1.17549435082 * 10-38 to ✥6.80564774407 * 1038 |
| long double | 4 | ✥1.17549435082 * 10-38 to ✥6.80564774407 * 1038 |

**Enumerations**

An enumeration data type is used for representing an abstract, discreet set of values with appropriate symbolic names.

**Enumeration Declaration**

Enumeration is declared like this:

```
enumtag {enumeration-list};
```

Here, tag is an optional name of the enumeration; enumeration-list is a comma-delimited list of discreet values, enumerators (or enumeration constants). Each enumerator is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator is set to zero, and each succeeding enumerator is set to one more than its predecessor.

Variables of enum type are declared same as variables of any other type. For example, the following declaration:

```
enumcolors { black, red, green, blue, violet, white } c;
```

establishes a unique integral type, enumcolors, a variable c of this type, and a set of enumerators with constant integer values (black = 0, red = 1, ...). In C, a variable of an enumerated type can be assigned any value of type int – no type checking beyond that is enforced. That is:

```
c = red;       // OK
c = 1;         // Also OK, means the same
```

With explicit integral initialisers, you can set one or more enumerators to specific values. The initialiser can be any expression yielding a positive or negative integer value (after possible integer promotions). Any subsequent names without initialisers will then increase by one. These values are usually unique, but duplicates are legal. The order of constants can be explicitly re-arranged. For example:

```
enumcolors { black,        // value 0
             red,          // value 1
             green,        // value 2
             blue=6,       // value 6
             violet,       // value 7
             white=4 };    // value 4
```

Initialiser expression can include previously declared enumerators. For example, in the following declaration:

```
enummemory_sizes { bit = 1, nibble = 4 * bit, byte = 2
* nibble,
                             kilobyte = 1024 * byte };
```

nibble would acquire the value 4, byte the value 8, and kilobyte the value 8192.

## Anonymous Enum Type

In our previous declaration, the identifier colors is the optional enumeration tag that can be used in subsequent declarations of enumeration variables of type enumcolors:

```
enumcolorsbg, border;   /* declare variables bg and
border */
```

As with struct and union declarations, you can omit the tag if no further variables of this enum type are required:

```
/* Anonymous enum type: */
enum { black, red, green, blue, violet, white } color;
```

## Enumeration Scope

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers:

```
int blue = 73;
{ // open a block
```

```
  enumcolors { black, red, green, blue, violet, white } c;
      /* enumerator blue = 3 now hides outer declaration
of int blue */
  structcolors { int i, j; };      // ILLEGAL: colors
duplicate tag
      double red = 2;      // ILLEGAL: redefinition of red
  }
  blue = 37;                      // back in int blue scope
```

**Void Type**

void is a special type indicating the absence of any value. There are no objects of void; instead, void is used for deriving more complex types.

**Void Functions**

Use the void keyword as a function return type if the function does not return a value.

```
  void print_temp(char temp) {
  Lcd_Out_Cp("Temperature:");
  Lcd_Out_Cp(temp);
  Lcd_Chr_Cp(223);  // degree character
  Lcd_Chr_Cp('C');
  }
```

Use void as a function heading if the function does not take any parameters. Alternatively, you can just write empty parentheses:

```
  main(void) { // same as main()
    ...
  }
```

## 4.2.2 Generic Pointers

Pointers can be declared as void, meaning that they can point to any type. These pointers are sometimes called generic.

**Derived Types**

The derived types are also known as structured types. These types are used as elements in creating more complex user-defined types. The derived types include arrays, pointers, structures, unions, arrays.

Array is the simplest and most commonly used structured type. Variable of array type is actually an array of objects of the same type. These objects represent elements of an array and are identified by their position in array. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.

### 4.2.3 Array declaration

Array declaration is similar to variable declaration, with the brackets added after identifer:

```
typearray_name[constant-expression]
```

This declares an array named as array_name composed of elements of type. The type can be scalar type (except void), user-defined type, pointer, enumeration, or another array. Result of the constant-expression within the brackets determines the number of elements in array. If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array.

Each of the elements of an array is numbered from 0 through the number of elements minus one. If the number is n, elements of array can be approached as variables array_name[0] .. array_name[n-1] of type.

Here are a few examples of array declaration:

```
#define MAX = 50
intvector_one[10];/* declares an array of 10 integers */
float   vector_two[MAX];            /* declares an array
of 50 floats    */
float  vector_three[MAX - 20];  /* declares an array of
30 floats    */
```

### Array initialisation

Array can be initialised in declaration by assigning it a comma-delimited sequence of values within braces. When initializing an array in declaration, you can omit the number of elements – it will be automatically determined according to the number of elements assigned. For example:

```
/* Declare an array which holds number of days in each
month: */
int days[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
/* This declaration is identical to the previous one */
int days[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

If you specify both the length and starting values, the number of starting values must not exceed the specified length. Vice versa is possible, when the trailing "excess" elements will be assigned some encountered runtime values from memory.

In case of array of char, you can use a shorter string literal notation. For example:

```
/* The two declarations are identical: */
const char msg1[] = {'T', 'e', 's', 't', '\0'};
const char msg2[] = "Test";
```

**Arrays in Expressions**

When name of the array comes up in expression evaluation (except with operators & and sizeof ), it is implicitly converted to the pointer pointing to array's first element.

**Multi-dimensional Arrays**

An array is one-dimensional if it is of scalar type. One-dimensional arrays are sometimes referred to as vectors. Multidimensional arrays are constructed by declaring arrays of array type. These arrays are stored in memory in such way that the right most subscript changes fastest, i.e. arrays are stored "in rows". Here is a sample 2-dimensional array:

```
  float m[50][20];   /* 2-dimensional array of size 50x20
*/
```

Variable m is an array of 50 elements, which in turn are arrays of 20 floats each. Thus, we have a matrix of 50x20 elements: the first element is m[0][0], the last one is m[49][19]. First element of the 5th row would be m[0][5].

If you are not initializing the array in the declaration, you can omit the first dimension of multi-dimensional array. In that case, array is located elsewhere, e.g. in another file. This is a commonly used technique when passing arrays as function parameters:

```
  int a[3][2][4];  /* 3-dimensional array of size 3x2x4 */
  voidfunc(int n[][2][4]) { /* we can omit first dimension
*/
    //...
    n[2][1][3]++;  /* increment the last element*/
  }//~
  void main() {
    //...
  func(a);
}//~!
```

You can initialise a multi-dimensional array with an appropriate set of values within braces. For example:

```
int a[3][2] = {{1,2}, {2,6}, {3,7}};
```

### 4.2.4 Pointers

Pointers are special objects for holding (or "pointing to") memory addresses. In C, address of an object in memory can be obtained by means of unary operator &. To reach the pointed object, we use indirection operator (*) on a pointer.

A pointer of type "pointer to object of type" holds the address of (that is, points to) an object of type. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, and unions.

A pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called.

Although pointers contain numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for declarations, assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

#### Pointer declarations

Pointers are declared same as any other variable, but with * ahead of identifier. Type at the beginning of declaration specifies the type of a pointed object. A pointer must be declared as pointing to some particular type, even if that type is void, which really means a pointer to anything. Pointers to void are often called generic pointers, and are treated as pointers to char in CCS.

If type is any predefined or user-defined type, including void, the declaration `type *p;     /* Uninitialized pointer */` declares p to be of type "pointer to type". All the scoping, duration, and visibility rules apply to the p object just declared. You can view the declaration in this way: if *p is an object of type, then p has to be a pointer to such objects.

Note: You must initialise pointers before using them! Our previously declared pointer *p is not initialized (i.e. assigned a value), so it cannot be used yet. In case of multiple pointer declarations, each identifier requires an indirect operator.

For example:

```
int *pa, *pb, *pc;
/* is same as: */
int *pa;
int *pb;
int *pc;
```

Once declared, a pointer can usually be reassigned so that it points to an object of another type. CCS lets you reassign pointers without typecasting, but the compiler will warn you unless the pointer was originally declared to be pointing to void. You can assign a void* pointer to a non-void* pointer – refer to void for details.

**Null pointers**

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it. Instead of zero, the mnemonic NULL (defined in the standard library header files, such as stdio.h) can be used for legibility. All pointers can be successfully tested for equality or inequality to NULL.

For example:

```
int *pn = 0;       /* Here's one null pointer */
int *pn = NULL;   /* This is an equivalent declaration */

/* We can test the pointer like this: */
if ( pn == 0 ) { ... }

/* .. or like this: */
if ( pn == NULL ) { ... }
```

The pointer type “pointer to void” must not be confused with the null pointer. The declaration `void *vp;` declares that vp is a generic pointer capable of being assigned to by any “pointer to type” value, including null, without complaint.

Assignments without proper casting between a “pointer to type1” and a “pointer to type2”, where type1 and type2 are different types, can invoke a compiler warning or error. If type1 is a function and type2 isn’t (or vice versa), pointer assignments are illegal. If type1 is a pointer to void, no cast is needed. If type2 is a pointer to void, no cast is needed.

### Function pointers

Function pointers are pointers, i.e. variables, which point to the address of a function.

```
// Define a function pointer
int (*pt2Function) (float, char, char);
```

Note: Thus functions and function pointers with different calling convention (argument order, arguments type or return type is different) are incompatible with each other.

### Assign an address to a Function Pointer

It's quite easy to assign the address of a function to a function pointer. You simply take the name of a suitable and known function or member function. It's optional to use the address operator & infront of the function's name.

```
//Assign an address to the function pointer
intDoIt  (float a, char b, char c){ return a+b+c; }
  pt2Function = &DoIt;  // assignment
```

Example:

```
intaddC(charx,char y){
returnx+y;
}
intsubC(charx,char y){
return x-y;
}
intmulC(charx,char y){
return x*y;
}
intdivC(charx,char y){
return x/y;
}
intmodC(charx,char y){
returnx%y;
}

//array of pointer to functions that receive two chars
and returns int
int (*arrpf[])(char,char) = { addC ,subC,mulC,divC,modC};
int res;
char i;
void main() {
for (i=0;i<5;i++){
     res = arrpf[i](10,20);
   }
}//~!
```

### 4.2.5 Structures

A structure is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure member can be a bit field type not allowed elsewhere.

Unlike arrays, structures are considered to be single objects. The CCS structure type lets you handle complex data structures almost as easily as single variables. CCS does not support anonymous structures (ANSI divergence).

#### Structure declaration and initialisation

Structures are declared using the keyword struct:

```
structtag {member-declarator-list};
```

Here, tag is the name of the structure; member-declarator-list is a list of structure members, actually a list of variable declarations. Variables of structured type are declared same as variables of any other type.

The member type cannot be the same as the struct type being currently declared. However, a member can be a pointer to the structure being declared, as in the following example:

```
structmystruct {mystruct s;};    /* illegal! */
structmystruct {mystruct *ps;}; /* OK */
```

Also, a structure can contain previously defined structure types when declaring an instance of a declared structure. Here is an example:

```
/* Structure defining a dot: */
struct Dot {float x, y;};
/* Structure defining a circle: */
struct Circle {
float r;
struct Dot center;
} o1, o2;
/* declare variables o1 and o2 of Circle */
```

Note that you can omit structure tag, but then you cannot declare additional objects of this type elsewhere. For more information, see the "Untagged Structures" below.

Structure is initialised by assigning it a comma-delimited sequence of values within braces, similar to array. For example:

```
/* Referring to declarations from the example above: */
/* Declare and initialize dots p and q: */
struct Dot p = {1., 1.}, q = {3.7, -0.5};
/* Declare and initialize circle o1: */
struct Circle o1 = {1., {0., 0.}}; // radius is 1,
center is at (0, 0)
```

### Incomplete declarations

Incomplete declarations are also known as forward declarations. A pointer to a structure type A can legally appear in the declaration of another structure B before A has been declared:

```
struct A;  // incomplete
struct B {struct A *pa;};
struct A {struct B *pb;};
```

The first appearance of A is called incomplete because there is no definition for it at that point. An incomplete declaration is allowed here, because the definition of B doesn't need the size of A.

### Untagged structures and typedefs

If you omit the structure tag, you get an untagged structure. You can use untagged structures to declare the identifiers in the comma-delimited member-declarator-list to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere.

It is possible to create a typedef while declaring a structure, with or without a tag:

```
/* With tag: */
typedefstructmystruct { ... } Mystruct;
Mystruct s, *ps, arrs[10];  /* same as structmystruct
s, etc. */
/* Without tag: */
typedefstruct { ... } Mystruct;
```

```
Mystruct s, *ps, arrs[10];
```

Usually, you don't need both tag and typedef: either can be used in structure type declarations.

Untagged structure and union members are ignored during initialisation.

### 4.2.6 Types conversions

C is strictly typed language, with each operator, statement and function demanding appropriately typed operands/arguments. However, we often have to use objects of "mismatching" types in expressions. In that case, type conversion is needed.

Conversion of object of one type is changing it to the same object of another type (i.e. applying another type to a given object). C defines a set of standard conversions for built-in types, provided by compiler when necessary

Conversion is required in following situations:

if statement requires an expression of particular type (according to language definition), and we use an expression of different type,

if operator requires an operand of particular type, and we use an operand of different type,

if a function requires a formal parameter of particular type, and we pass it an object of different type,

if an expression following the keyword return does not match the declared function return type,

In these situations, compiler will provide an automatic implicit conversion of types, without any user interference. Also, user can demand conversion explicitly by means of typecast operator.

### 4.2.7 Declarations

Declaration introduces one or several names to a program – it informs the compiler what the name represents, what is its type, what are allowed operations with it, etc. This section reviews concepts related to declarations: declarations, definitions, declaration specifiers, and initialization.

The range of objects that can be declared includes:

Variables

Constants

Functions

Types

Structure, union, and enumeration tags

Structure members

Union members
Arrays of other types
Statement labels
Pre-processor macros

### Declarations and Definitions

Defining declarations, also known as definitions, besides introducing the name of an object, also establish the creation (where and when) of the object; that is, the allocation of physical memory and its possible initialization. Referencing declarations, or just declarations, simply make their identifiers and types known to the compiler.

Here is an overview. Declaration is also a definition, except if:

it declares a function without specifying its body

it has an extern specifier, and has no initializator or body (in case of function)

it is a typedef declaration

There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

For example:

```
/* Here is a nondefining declaration of function max; */
/* it merely informs compiler that max is a function */
int max();

/* Here is a definition of function max: */
int max(int x, int y) {
return (x >= y) ? x : y;
}

/* Definition of variable i: */
inti;

/* Following line is an error, i is already defined! */
inti;
```

### Declarations and Declarators

A declaration is a list of names. The names are sometimes referred to as declarators or identifiers. The declaration begins with optional storage class specifiers, type specifiers, and other modifiers. The identifiers are separated by commas and the list is terminated by a semicolon.

Declarations of variable identifiers have the following pattern:

```
  storage-class [type-qualifier] typevar1 [=init1], var2
[=init2], ... ;
```

where var1, var2,... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of type; if omitted, type defaults to int. Specifierstorage-class can take values extern, static, register, or the default auto. Optional type-qualifier can take values constant or volatile.

For example:

```
  /* Create 3 integer variables called x, y, and z
       and initialize x and y to the values 1 and 2,
respectively: */
  int x = 1, y = 2, z;    // z remains uninitialized

  /* Create a floating-point variable q with static
modifier,
      and initialize it to 0.25: */
  static float q = .25;
```

These are all defining declarations; storage is allocated and any optional initializers are applied.

### 4.2.8 Functions

Functions are central to C programming. Functions are usually defined as subprograms which return a value based on a number of input parameters. Return value of a function can be used in expressions – technically, function call is considered to be an expression like any other.

C allows a function to create results other than its return value, referred to as side effects. Often, function return value is not used at all, depending on the side effects. These functions are equivalent to procedures of other programming languages, such as Pascal. C does not distinguish between procedure and function – functions play both roles.

Each program must have a single external function named main marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions have external linkage by default and are normally accessible from any file in the program. This can be restricted by using the static storage class specifier in function declaration.

### Function Declaration

Functions are declared in your source files or made available by linking precompiled libraries. Declaration syntax of a function is:

```
typefunction_name(parameter-declarator-list);
```

The function_name must be a valid identifier. This name is used to call the function.

The type represents the type of function result, and can be any standard or user-defined type. For functions that do not return value, you should use void type. The type can be omitted in global function declarations, and function will assume int type by default.

Function type can also be a pointer. For example, float* means that the function result is a pointer to float. Generic pointer, void* is also allowed.

Function cannot return an array or another function.

Within parentheses, parameter-declarator-list is a list of formal arguments that function takes. These declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. If the list is empty, function does not take any arguments. Also, if the list is void, function also does not take any arguments; note that this is the only case when void can be used as an argument's type.

Unlike with variable declaration, each argument in the list needs its own type specifier and a possible qualifier const or volatile.

### Function Prototypes

A given function can be defined only once in a program, but can be declared several times, provided the declarations are compatible. If you write a nondefining declaration of a function, i.e. without the function body, you do not have to specify the formal arguments.

This kind of declaration, commonly known as the function prototype, allows better control over argument number and type checking, and type conversions. Name of the parameter in function prototype has its scope limited to the prototype. This allows different parameter names in different declarations of the same function:

```
  /* Here are two prototypes of the same function: */

  int test(const char*)    /* declares function test */
  int test(const char*p)   /* declares the same function
test */
```

Function prototypes greatly aid in documenting code. For example, the function Cf_Init takes two parameters: Control Port and Data Port. The question is, which is which? The function prototype

```
  voidCf_Init(char *ctrlport, char *dataport);
```

makes it clear. If a header file contains function prototypes, you can that file to get the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is used only for any later error messages involving that parameter; it has no other effect.

**Function Definition**

Function definition consists of its declaration and a function body. The function body is technically a block – a sequence of local definitions and statements enclosed within braces {}. All variables declared within function body are local to the function, i.e. they have function scope.

The function itself can be defined only within the file scope. This means that function declarations cannot be nested.

To return the function result, use the return statement. Statement return in functions of void type cannot have a parameter – in fact, you can omit the return statement altogether if it is the last statement in the function body.

Here is a sample function definition:

```
  /* function max returns greater one of its 2 arguments:
*/
```

```
int max(int x, int y) {
return (x>=y) ? x : y;
}
```

Here is a sample function which depends on side effects rather than return value:

```
/*simpleFunc.c -- to demonstrate function calls*/
#INCLUDE <16F873.h>
#USE DELAY (CLOCK=4000000)
short int RB0toRC0(RC1val)
short int RC1val;
{
Output_bit(pin_C1, RC1val);  /*Set RC1 to the specified
value*/
    if (input(pin_B0)) {          /*Read RB0*/
Output_high(pin_C0);          /*If RB0 is high, RC0 set
high*/
    }
    else {
Output_low(pin_C0);          /*else set RC0 low*/
    }
    return(input(pin_B0));          /*Return RB0*/
}
void main() {
    short int b;

    b=RB0toRC0(1);
    b=RB0toRC0(0);
}
```

## Functions reentrancy

Limited reentrancy for functions is allowed. The functions that don't have their own function frame (no arguments and local variables) can be called both from the interrupt and the "main" thread.

Functions that have input arguments and/or local variables can be called only from one of the before mentioned program threads.

### 4.2.8 Operators

C has a variety of built in operations for performing math. These are listed below along with an example where

a=0x03, and b=0x11:

| | Name of Operand | Symbol | Example | Result a=0x03 b=0x11 |
|---|---|---|---|---|
| Binary Operators (Two Operands) | Addition | + | a+b | 0x14 |
| | Subtraction | - | b-a | 0x0E |
| | Multiplication | * | a*b | 0x33 |
| | Division | / | b/a | 0x05 |
| | Modulus | % | b%a | 0x02 |
| | (remainder) | | | |
| | Bitwise and | & | b&a | 0x01 |
| | Bitwise or | \| | b\|a | 0x13 |
| | Bitwise xor | ^ | b^a | 0x12 |
| | Shift right | >> | b>>a | 0x02 |
| | Shift left | << | b<<a | 0x88 |
| Unary Operators (One Operand) | increment | ++ | ++a | 0x04 |
| | decrement | -- | --a | 0x03 |
| | negate | - | -a | -0x03 |
| | logical complement | ~ | ~a | 0xFC |

**Logical Expressions**

In addition to manipulating numbers, C is also capable of handling logical expressions. When using these expressions TRUE is taken to be anything that is not zero, and FALSE is always equal to zero. Again, a=0x03,

and b=0x11:

Binary operators (two operands)

| | Name of Operand | Symbol | Example | Result a=0x03 b=0x11 |
|---|---|---|---|---|
| Binary Operators | Greater than | > | a>b | FALSE |
| | Less than | < | a<b | TRUE |
| | Equal | == | a==b | FALSE |
| | Greater than or equal | >= | a>=b | FALSE |
| | Less than or equal | <= | a<=b | TRUE |
| | Not equal | != | a!=b | TRUE |
| | Logical AND | && | a&&b | TRUE |
| | Logical OR | \|\| | a\|\|b | TRUE |
| Unary operators (one operand) | Logical complement | ! | !a | FALSE |

```
Example for use of operators:
while (a!=b && a<=c){
++a;
--b;
/* compute something more… */
}
```

### 4.2.9 Statements

Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. C has a variety of mechanisms to control the flow of a program. These are listed below:

```
   The if...then construct
   if (logical expression) {
      ...statements...
}
```

If the logical expression is true then evaluate the statements between the braces. The following code sets RC1 if a is even.

```
  if ((a%2) == 0) {
   Output_high(pin_C1);
}

  The if...then...else construct
  if (logical expression) {
   ...if statements...
}
else {
   ...else statements...
}
```

If the logical expression is true then evaluate the "if" statements between the braces, otherwise execute the "else" statements. The following code decides if a number if is even or odd.

```
  if ((a%2) == 0) {
   Output_high(pin_C1);
}
else {
   Output_low(pin_C1);
}

  The while loop
  while (logical expression) {
       ...statements...
  }
```

While the logical expression is true, the statements (of which there is an arbitrary number) between the braces is executed. The following code cycles through the even numbers from 0 to 9 (the variables must have been declared elsewhere in the program).

```
  a=0;
  while (a<10) {
       ...statements...
       a=a+2;
  }

  The for loop
  for (initial condition; logical expression; change
loop counter variable) {
    ...statements...
}
```

Set the initial condition, then while the logical expression is true execute the statement between the braces while changing the loop counter as specified once at the end of each loop. This code segment is functionally equivalent to the one for the "while" loop.

```
  for (a=0; a<10; a=a+2) {
    ...statements...
}
```

**The case...switch construct**

Case..switch is used in place of a series of "if...else" clauses when a variable can take on several values.  The "default" clause at the bottom takes care of any cases not covered explicitly.

```
  switch (variable) {
   case val1: ...statements 1...
   break;
     case val2: ...statements 2...
   break;
     case val3: ...statements 3...
   break;
     default: ...statements default...
   break;
  }
```

## 4.2.10 PRE-PROCESSOR

Pre-processor directives all begin with a # and are followed by a specific command.  Syntax is dependent on the command. Many commands do not allow other syntactical elements on the remainder of the line.  A table of commands and a description is listed on the below:

We will now explain some of the common preprocessor directives used in PIC C programs:

| | | | |
|---|---|---|---|
| Standard C | #DEFINE ID STRING | #IF expr | #NOLIST |
| | #ELSE | #IFDEF id | #PRAGMA cmd |
| | #ENDIF | #INCLUDE "FILENAME" | #UNDEF id |
| | #ERROR | #LIST | |
| Function Qualifier | #INLINE | #INT_GLOBAL | #SEPARATE |
| | #INT_DEFAULT | #INT_xxx | |
| Pre-Defined Identifier | __DATE__ | __LINE__ | __PCH__ |
| | __DEVICE__ | __PCB__ | __TIME__ |
| | __FILE__ | __PCM__ | __FILENAME__ |
| RTOS | #TASK | #USE RTOS | |
| Device Specification | #DEVICE CHIP | #ID "filename" | |
| | #FUSES options | #ID NUMBER | |
| | #ID CHECKSUM | #SERIALIZE | |
| Built-in Libraries | #USE DELAY CLOCK | #USE FIXED_IO | #USE RS232 |
| | #USE FAST_IO | #USE I2C | #USE STANDARD_IO |
| | #USE SPI | | |
| Memory Control | #ASM | #BYTE id=id | #ROM |
| | #BIT id=id.const | #ENDASM | #TYPE |
| | #BIT id=const.const | #FILL_ROM | #ZERO_RAM |
| | #BUILD | #LOCATE id=const | |
| | #BYTE id=const | #RESERVE | |
| Compiler Control | #CASE | #OPT n | #PRIORITY |
| | #IGNORE_WARNINGS | #ORG | |

**Fuses**

| | |
|---|---|
| Syntax: | #fuse options |
| Elements: | options vary depending on the device. A list of all valid options is always present at the top of each devices .h file in a comment for reference. The CCS PCW device edit utility can modify a particular devices fuses. The PCW pull down menu VIEW \| Valid fuses will show all fuses with their descriptions.<br><br>Some common options are:<br>LP, XT, HS, RC<br>WDT, NOWDT<br>PROTECT, NOPROTECT<br>PUT, NOPUT (Power Up Timer)<br>BROWNOUT, NOBROWNOUT |
| Purpose: | This directive defines what fuses should be set in the part when it is programmed. This directive does not affect the compilation; however, the information is put in the output files. If the fuses need to be in Parallax format, add a PAR option. SWAP has the special function of swapping (from the Microchip standard) the high and low BYTES of non-program data in the Hex file. This is required for some device programmers.<br><br>Some processors allow different levels for certain fuses. To access these levels, assign a value to the fuse. For example, on the 18F452, the fuse PROTECT=6 would place the value 6 into CONFIG5L, protecting code blocks 0 and 3.<br><br>When linking multiple compilation units be aware this directive applies to the final object file. Later files in the import list may reverse settings in previous files. |
| Examples: | #fuses HS,NOWDT |

```
#DEFINE

Syntax:
#define id text
   or
#define id(x,y...)  text
```

**Elements:**
id is a preprocessor identifier, text is any text, x,y and so on are local preprocessor identifiers, and in this form there may be one or more identifiers separated by commas.

**Purpose:**
Used to provide a simple string replacement of the ID with the given text from this point of the program and on.

In the second form (a C macro) the local identifiers are matched up with similar identifiers in the text and they are replaced with text passed to the macro where it is used.

If the text contains a string of the form #idx then the result upon evaluation will be the parameter id concatenated with the string x.

If the text contains a string of the form #idx#idy then parameter idx is concatenated with parameter idy forming a new identifier.

Examples:

```
#define  BITS  8
a=a+BITS;   //same as   a=a+8;


#define hi(x)  (x<<4)
a=hi(a);    //same as   a=(a<<4);
```

## #INCLUDE

| | |
|---|---|
| Syntax: | #include <filename> or #include "filename" |
| Elements: | filename is a valid PC filename. It may include normal drive and path information. A file with the extension ".encrypted" is a valid PC file. The standard compiler #include directive will accept files with this extension and decrypt them as they are read. This allows include files to be distributed without releasing the source code. |
| Purpose: | Text from the specified file is used at this point of the compilation. If a full path is not specified the compiler will use the list of directories specified for the project to search for the file. If the filename is in "" then the directory with the main source file is searched first. If the filename is in <> then the directory with the main source file is searched last. |
| Examples: | `#include  <16C54.H>`<br>`#include  <C:\INCLUDES\COMLIB\MYRS232.C>` |

## #USE DELAY

| | |
|---|---|
| Syntax: | #use delay (clock=speed)<br>#use delay (clock=speed, restart_wdt)<br>#use delay (clock=speed, type)<br>#use delay (clock=speed, type=speed)<br>#use delay (type=speed) |
| Elements: | speed is a constant 1-100000000 (1 hz to 100 mhz). This number can contains commas. This number also supports the following denominations: M, MHZ, K, KHZ<br>type defines what kind of clock you are using, and the following values are valid: oscillator, osc (same as oscillator), crystal, xtal (same as crystal), internal, int (same as internal) or rc. The compiler will automatically set the oscillator configuration bits based upon your defined type. If you specified internal, the compiler will also automatically set the internal oscillator to the defined speed.<br>restart_wdt will restart the watchdog timer on every delay_us() and delay_ms() use. |
| Purpose: | Tells the compiler the speed of the processor and enables the use of the built-in functions: delay_ms() and delay_us(). Will also set the proper configuration bits, and if needed configure the internal oscillator. Speed is in cycles per second. An optional restart_WDT may be used to cause the compiler to restart the WDT while delaying. When linking multiple compilation units, this directive must appear in any unit that needs timing configured (delay_ms(), delay_us(), UART, SPI). In multiple clock speed applications, this directive may be used more than once. Any timing routines (delay_ms(), delay_us, UART, SPI) that need timing information will use the last defined #use delay(). For initialization purposes, the compiler will initialize the configuration bits and internal oscillator based upon the first #use delay(). |

**Video!**
On this month's DVD, we've included an exciting video of our robot in the making. You'll find the step-by-step instructions easy to follow, especially if you are not familiar with electronics and programming. Write to ***editor@thinkdigit.com*** and tell us about your experience.

| | |
|---|---|
| Examples: | <pre>/set timing config to 32KHz, restart watchdog timer<br>//on delay_us() and delay_ms()<br>#use delay (clock=32000, RESTART_WDT)<br><br>//the following 4 examples all configure the timing library<br>//to use a 20Mhz clock, where the source is an oscillator.<br>#use delay (clock=20000000)          //user must manually set HS config bit<br>#use delay (clock=20,000,000)        //user must manually set HS config bit<br>#use delay(clock=20M)                //user must manually set HS config bit<br>#use delay(clock=20M, oscillator)    //compiler will set HS config bit<br>#use delay(oscillator=20M)           //compiler will set HS config bit<br><br>//application is using a 10Mhz oscillator, but using the 4x PLL<br>//to upscale it to 40Mhz.  Compiler will set H4 config bit.<br>#use delay(clock=40M, oscillator=10M)<br><br>//application will use the internal oscillator at 8MHz.<br>//compiler will set INTOSC_IO config bit, and set the internal<br>//oscillator to 8MHz.<br>#use delay(internal=8M)</pre> |

Note: You can find out more about other pre-processor directives by visiting the links mentioned in the reference section.

## 2.11. BUILT-IN-FUNCTIONS

The CCS compiler provides a lot of built-in functions to access and use the pic microcontroller's peripherals. This makes it very easy for the users to configure and use the peripherals without going into in depth details of the registers associated with the functionality. The functions categorized by the peripherals associated with them are listed on the next page. Click on the function name to get a complete description and parameter and return value descriptions.

| | | | | |
|---|---|---|---|---|
| RS232 I/O | ASSERT( ) | GETCH( ) | **PUTC( )** | |
| | FGETC( ) | GETCHAR( ) | **PUTCHAR( )** | |
| | FGETS( ) | GETS( ) | **PUTS( )** | |
| | FPRINTF( ) | KBHIT( ) | **SET_UART_ SPEED( )** | |
| | FPUTC( ) | PERROR( ) | **SETUP_UART( )** | |
| | FPUTS( ) | **PRINTF( )** | | |
| | | | | |
| SPI TWO WIRE I/O | SETUP_SPI( ) | SPI_DATA_IS_IN( ) | SPI_READ( ) | **SPI_WRITE( )** |
| | **SPI_XFER( )** | | | |
| | | | | |
| DISCRETE I/O | GET_TRISx( ) | INPUT_K( ) | OUTPUT_FLOAT( ) | **SET_TRIS_B( )** |
| | INPUT( ) | INPUT_STATE( ) | OUTPUT_G( ) | **SET_TRIS_C( )** |
| | INPUT_A( ) | INPUT_x( ) | OUTPUT_H( ) | **SET_TRIS_D( )** |
| | INPUT_B( ) | OUTPUT_A( ) | OUTPUT_HIGH( ) | **SET_TRIS_E( )** |
| | INPUT_C( ) | OUTPUT_B( ) | OUTPUT_J( ) | **SET_TRIS_F( )** |
| | INPUT_D( ) | OUTPUT_BIT( ) | OUTPUT_K( ) | **SET_TRIS_G( )** |
| | INPUT_E( ) | OUTPUT_C( ) | OUTPUT_LOW( ) | **SET_TRIS_H( )** |
| | INPUT_F( ) | OUTPUT_D( ) | OUTPUT_TOG-GLE( ) | **SET_TRIS_J( )** |
| | INPUT_G( ) | OUTPUT_DRIVE( ) | PORT_A_PUL-LUPS() | **SET_TRIS_K( )** |
| | INPUT_H( ) | OUTPUT_E( ) | **PORT_B_PUL-LUPS()** | |
| | INPUT_J( ) | OUTPUT_F( ) | **SET_TRIS_A( )** | |
| | | | | |
| PARALLEL SLAVE I/O | PSP_INPUT_FULL( ) | **PSP_OVERFLOW( )** | | |
| | PSP_OUTPUT_FULL( ) | **SETUP_PSP( )** | | |
| | | | | |

| | | | | |
|---|---|---|---|---|
| I2C I/O | I2C_ISR_STATE() | I2C_SlaveAddr( ) | **I2C_WRITE( )** | |
| | I2C_POLL( ) | **I2C_START( )** | | |
| | I2C_READ( ) | **I2C_STOP( )** | | |
| | | | | |
| PROCESSOR CONTROLS | CLEAR_INTERRUPT( ) | GOTO_ADDRESS( ) | **RESET_CPU( )** | |
| | DISABLE_INTERRUPTS( ) | INTERRUPT_AC-TIVE( ) | **RESTART_ CAUSE( )** | |
| | ENABLE_INTERRUPTS( ) | JUMP_TO_ISR | **SETUP_OSCIL-LATOR( )** | |
| | EXT_INT_EDGE( ) | LABEL_ADDRESS( ) | **SLEEP( )** | |
| | GETENV( ) | READ_BANK( ) | **WRITE_BANK( )** | |
| | | | | |
| BIT/BYTE MANIPU-LATION | BIT_CLEAR( ) | MAKE8( ) | _MUL( ) | **SHIFT_LEFT( )** |
| | BIT_SET( ) | MAKE16( ) | ROTATE_LEFT( ) | **SHIFT_RIGHT( )** |
| | BIT_TEST( ) | MAKE32( ) | ROTATE_RIGHT( ) | **SWAP( )** |
| | | | | |
| STANDARD C MATH | ABS( ) | COSH( ) | LABS( ) | **SIN( )** |
| | ACOS( ) | DIV( ) | LDEXP( ) | **SINH( )** |
| | ASIN( ) | EXP( ) | LDIV( ) | **SQRT( )** |
| | ATAN( ) | FABS( ) | LOG( ) | **TAN( )** |
| | ATAN2( ) | FLOOR( ) | LOG10( ) | **TANH( )** |
| | CEIL( ) | FMOD( ) | **MODF( )** | |
| | COS( ) | FREXP( ) | **POW( )** | |
| | | | | |
| VOLTAGE REF | SETUP_LOW_VOLT_ DETECT( ) | **SETUP_VREF( )** | | |
| | | | | |

| | | | | |
|---|---|---|---|---|
| A/D CONVERSION | SET_ADC_CHANNEL( ) | **SETUP_ADC_ PORTS( )** | | |
| | SETUP_ADC( ) | **READ_ADC( )** | | |
| | | | | |
| STANDARD C CHAR | ATOF( ) | ISLOWER(char) | STRCMP( ) | **STRRCHR( )** |
| | ATOI( ) | ISPRINT(x) | STRCOLL( ) | **STRSPN( )** |
| | ATOI32( ) | ISPUNCT(x) | STRCPY( ) | **STRSTR( )** |
| | ATOL( ) | ISSPACE(char) | STRCSPN( ) | **STRTOD( )** |
| | ISALNUM( ) | ISUPPER(char) | STRLEN( ) | **STRTOK( )** |
| | ISALPHA(char) | ISXDIGIT(char) | STRLWR( ) | **STRTOL( )** |
| | ISAMOUNG( ) | ITOA( ) | STRNCAT( ) | **STRTOUL( )** |
| | ISCNTRL(x) | SPRINTF( ) | STRNCMP( ) | **STRXFRM( )** |
| | ISDIGIT(char) | STRCAT( ) | STRNCPY( ) | **TOLOWER( )** |
| | ISGRAPH(x) | STRCHR( ) | STRPBRK( ) | **TOUPPER( )** |
| | | | | |
| TIMERS | GET_TIMER0( ) | SET_RTCC( ) | **SETUP_TIM-ER_0 ( )** | |
| | GET_TIMER1( ) | SET_TIMER0( ) | **SETUP_TIMER_1 ( )** | |
| | GET_TIMER2( ) | SET_TIMER1( ) | **SETUP_TIM-ER_2 ( )** | |
| | GET_TIMER3( ) | SET_TIMER2( ) | **SETUP_TIM-ER_3 ( )** | |
| | GET_TIMER4( ) | SET_TIMER3( ) | **SETUP_TIM-ER_4 ( )** | |
| | GET_TIMER5( ) | SET_TIMER4( ) | **SETUP_TIMER_5 ( )** | |
| | GET_TIMERx( ) | SET_TIMER5( ) | **SETUP_WDT ( )** | |
| | RESTART_WDT( ) | **SETUP_COUNTERS( )** | | |
| | | | | |

| | | | | |
|---|---|---|---|---|
| STANDARD C MEMORY | CALLOC( ) | MEMCMP( ) | **OFFSETOFBIT( )** | |
| | FREE( ) | MEMCPY( ) | **REALLOC( )** | |
| | LONGJMP( ) | MEMMOVE( ) | **SETJMP( )** | |
| | MALLOC( ) | **MEMSET( )** | | |
| | MEMCHR( ) | **OFFSETOF( )** | | |
| | | | | |
| STANDARD STRING | STRXFRM( ) | MEMCHR( ) | **MEMCMP( )** | |
| | STRCAT( ) | STRCHR( ) | **STRCMP( )** | |
| | STRCOLL( ) | STRCSPN( ) | **STRICMP( )** | |
| | STRCOLL( ) | STRCSPN( ) | **STRICMP( )** | |
| | STRLEN( ) | STRLWR( ) | **STRNCAT( )** | |
| | STRNCMP( ) | STRNCPY( ) | **STRPBRK( )** | |
| | STRRCHR( ) | STRSPN( ) | **STRSTR( )** | |
| | | | | |
| CAPTURE/COMPARE/PWM | SET_POWER_PWM_OVERRIDE( ) | **SETUP_CCP2( )** | | |
| | SET_POWER_PWMX_DUTY( ) | **SETUP_CCP3( )** | | |
| | SET_PWM1_DUTY( ) | **SETUP_CCP4( )** | | |
| | SET_PWM2_DUTY( ) | **SETUP_CCP5( )** | | |
| | SET_PWM3_DUTY( ) | **SETUP_CCP6( )** | | |
| | SET_PWM4_DUTY( ) | **SETUP_POWER_PWM( )** | | |
| | SET_PWM5_DUTY( ) | **SETUP_POWER_PWM_PINS( )** | | |
| | **SETUP_CCP1( )** | | | |
| | | | | |

| | | | | |
|---|---|---|---|---|
| INTERNAL EEPROM | ERASE_PROGRAM_EEP-ROM( ) | **SETUP_EXTER-NAL_MEMORY( )** | | |
| | READ_CALIBRATION( ) | **WRITE_CONFIGU-RATION_MEMORY( )** | | |
| | READ_EEPROM( ) | **WRITE_EEPROM( )** | | |
| | READ_EXTERNAL_MEM-ORY( ) | **WRITE_EXTER-NAL_MEMORY( )** | | |
| | READ_PROGRAM_EEP-ROM( ) | **WRITE_PROGRAM_EEPROM( )** | | |
| | READ_PROGRAM_MEM-ORY( ) | **WRITE_PROGRAM_MEMORY( )** | | |
| | | | | |
| STANDARD C SPECIAL | BSEARCH( ) | RAND( ) | SRAND( ) | **QSORT( )** |
| | | | | |
| DELAYS | DELAY_CYCLES( ) | DELAY_MS( ) | **DELAY_US( )** | |
| | | | | |
| ANALOG COM-PARE | **SETUP_COMPARA-TOR( )** | | | |
| | | | | |
| RTOS | RTOS_AWAIT( ) | RTOS_MSG_SEND( ) | **RTOS_TERMI-NATE( )** | |
| | RTOS_DISABLE( ) | RTOS_OVERRUN( ) | **RTOS_WAIT( )** | |
| | RTOS_ENABLE( ) | RTOS_RUN( ) | **RTOS_YIELD( )** | |
| | RTOS_MSG_POLL( ) | **RTOS_SIGNAL( )** | | |
| | RTOS_MSG_READ( ) | **RTOS_STATS( )** | | |
| | | | | |
| LCD | LCD_LOAD( ) | LCD_SYMBOL( ) | **SETUP_LCD( )** | |
| | | | | |
| MISC. | SETUP_OPAMP1( ) | SETUP_OPAMP2( ) | **SLEEP_UL-PWU( )** | |

As before, since basic line followers require only discrete I/O and delay functions, we are going to delve only in them. For information on other functions, visit the links mentioned in the reference section.

**INPUT ( )**

| | |
|---|---|
| Syntax: | value = input (pin) |
| Elements: | Pin to read. Pins are defined in the devices .h file. The actual value is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This is defined as follows: #define PIN_A3 43.<br>The PIN could also be a variable. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A.<br>Note that doing I/O with a variable instead of a constant will take much longer time. |
| Returns: | 0 (or FALSE) if the pin is low,<br>1 (or TRUE) if the pin is high |
| Function: | This function returns the state of the indicated pin. The method of I/O is dependent on the last USE *_IO directive. By default with standard I/O before the input is done the data direction is set to input. |
| Availability: | All devices. |
| Requires: | Pin constants are defined in the devices .h file |
| Examples: | `while ( !input(PIN_B1) );`<br>`// waits for B1 to go high`<br><br>`if( input(PIN_A0) )`<br>`   printf("A0 is now high\r\n");`<br><br>`int16 i=PIN_B1;`<br>`while(!i);`<br>`//waits for B1 to go high` |

### INPUT_x( )

| | |
|---|---|
| Syntax: | value = input_a()<br>value = input_b()<br>value = input_c()<br>value = input_d()<br>value = input_e()<br>value = input_f()<br>value = input_g()<br>value = input_h()<br>value = input_j()<br>value = input_k() |
| Parameters: | None |
| Returns: | An 8 bit int representing the port input data. |
| Function: | Inputs an entire byte from a port. The direction register is changed in accordance with the last specified #USE *_IO directive. By default with standard I/O before the input is done the data direction is set to input. |
| Availability: | All devices. |
| Requires: | Nothing. |
| Examples: | `int data;`<br>`data = input_b();`<br>`//if at port B input was suppose 0xf2`<br>`//then variable data's value is stored as 0xf2` |

### OUTPUT_A ( )

| | |
|---|---|
| Syntax: | output_a (value)<br>output_b (value)<br>output_c (value)<br>output_d (value)<br>output_e (value)<br>output_f (value)<br>output_g (value)<br>output_h (value)<br>output_j (value)<br>output_k (value) |
| Parameters: | value is a 8 bit int |
| Returns: | undefined |

| | |
|---|---|
| Function: | Output an entire byte to a port. The direction register is changed in accordance with the last specified #USE *_IO directive. |
| Availability: | All devices, however not all devices have all ports (A-E) |
| Requires: | Nothing. |
| Examples: | `OUTPUT_B(0xf0);`<br>`   // B7,B6,B5,B4 have 1 as output`<br>`   // B3,B2,B1,B0 have 0 as output` |

**SET_TRIS_A( )**

| | |
|---|---|
| Syntax: | set_tris_a (value)<br>set_tris_b (value)<br>set_tris_c (value)<br>set_tris_d (value)<br>set_tris_e (value)<br>set_tris_f (value)<br>set_tris_g (value)<br>set_tris_h (value)<br>set_tris_j (value)<br>set_tris_k (value) |
| Parameters: | value is an 8 bit int with each bit representing a bit of the I/O port. |
| Returns: | undefined |
| Function: | These functions allow the I/O port direction (TRI-State) registers to be set. This must be used with FAST_IO and when I/O ports are accessed as memory such as when a #BYTE directive is used to access an I/O port. Using the default standard I/O the built in functions set the I/O direction automatically.<br>Each bit in the value represents one pin. A 1 indicates the pin is input and a 0 indicates it is output. |
| Availability: | All devices (however not all devices have all I/O ports) |
| Requires: | Nothing. |
| Examples: | `SET_TRIS_B( 0x0F );`<br>`   // B7,B6,B5,B4 are outputs`<br>`   // B3,B2,B1,B0 are inputs` |

## DELAY_MS ( )

| | |
|---|---|
| Syntax: | delay_ms (time) |
| Parameters: | time - a variable 0-65535(int16) or a constant 0-65535<br>Note: Previous compiler versions ignored the upper byte of an int16, now the upper byte affects the time. |
| Returns: | undefined |
| Function: | This function will create code to perform a delay of the specified length. Time is specified in milliseconds. This function works by executing a precise number of instructions to cause the requested delay. It does not use any timers. If interrupts are enabled the time spent in an interrupt routine is not counted toward the time. The delay time may be longer than requested if an interrupt is serviced during the delay. The time spent in the ISR does not count toward the delay time. |
| Availability: | All devices |
| Requires: | #use delay |
| Examples: | `#use delay (clock=20000000)`<br><br>`delay_ms( 2 );`<br><br>`void delay_seconds(int n) {`<br>`  for (;n!=0; n- -)`<br>`  delay_ms( 1000 );` |

With this part, this section is concluded, if you read through all of it, you would have sufficient knowledge to code in C using CCS compiler for most kinds of PIC microcontrollers.

**Video!**
On this month's DVD, we've included an exciting video of our robot in the making. You'll find the step-by-step instructions easy to follow, especially if you are not familiar with electronics and programming. Write to ***editor@thinkdigit.com*** and tell us about your experience.

# 5 Algorithm and Logic

The line follower, in general can encounter three scenarios:

1. Straight path (line) ahead
2. Right turn ahead
3. Left turn ahead

Now, there is no problem when the path ahead is straight. The robot will keep on moving ahead until it encounters any bend or turn. When it encounters a turn, it has to change its direction of locomotion. When a turn towards left arises, it has to turn left and when a right turn is encountered it has to turn right. The direction of turn is recognised by sensors in front. When a left turn arises, the sensor S1 senses the colour of the surface beneath it changing from white to black while S2 is still white surface below it. This condition of sensors is different from the condition when the robot is on straight pat and both of its sensors, S1 and S2 are sensing white. Similarly, when a right turn is encountered, the S2 senses a black surface below it instead of white, while S1 continues to sense white surface. This helps the processor conclude that there is a right turn ahead and processor takes necessary steps. These conditions are tabulated below:

| S1 (Left sensor) | S2 (Right Sensor) | Conclusion |
|---|---|---|
| White | White | Straight path ahead |
| Black | White | Left turn ahead |
| White | Black | Right turn ahead |

The fourth condition, of both sensors sensing black, won't arise if all the angles in the path are acute. Including obtuse angles and really sharp turns in the path will ask for more efficient algorithm and code, and can be ignored for the time being.

One of the most important things to consider here is the positioning of the sensors. Since, we are following the detection scheme in which the white colour detection would mean the 'normal' condition and black colour detection would mean that there is a turn ahead, so we need to mount out sensors in such a way that they lie just outside the thickness of the black line. The correct way to mount the sensors will be discussed in the chapter 'Assembling and making it work'.

At this stage we have studied about the various options available to us in various aspects of programming, electronics and mechanics. Now we need to select what options we are going to use to build our robot. Following table gives the list of all that what we will use and their approximate cost in the market:

| Component Name | Category | Quantity | Cost Per Piece |
|---|---|---|---|
| IR transmitter | Sensor | 4 | 6 |
| IR receiver | Sensor | 4 | 6 |
| PIC 16F84 | Processing unit | 1 | 80 |
| LM358N | Opamp | 2 | 40 |
| L293D | Motor driver | 1 | 100 |
| DC Geared Motor | Actuators | 2 | 150 |
| 7805 | Voltage regulator | 1 | 5 |
| 7809 | Voltage regulator | 1 | 5 |
| Mica capacitors | Capacitor | 4 | 1 |
| Electrolytic capacitors | Capacitor | 4 | 5 |
| Potentiometer | Variable resistor | 1 | 5 |
| LED | Diode | 4 | 1 |
| PCB | Circuit building platform | 1 | 20 |
| Wires | Connections | 2 meters | 10 per meter |
| Batteries (1.5 V) | Power source | 8 | 8 |
| Breadboard | Circuit building platforms | 2 | 80 |

Apart from these, you will need a multimeter, soldering iron and wire (if you plan to make it on PCB). The total money which you would need to spend here is not much, considering the fact that almost all the components used here can be reused multiple number of times (One of the most important reason of using a breadboard here is to be able to reuse the components again ). So, take it as one time investment, as from now onwards you wont to spend again on tools like multimeter and soldering iron as well as components like microcontroller.

## 5.1 Brief tutorial on PIC

We will be using a PIC microcontroller for our project because of its versatility. So it will be good if we learn about PIC microcontrollers before we start working on them. This chapter provides basic knowledge which you would need to know before starting on with PICs.

PIC microcontrollers are a product from Microchip technology Inc. It's not clearly known whether PIC is an abbreviated form of something or not. Some people say it stands for Peripheral Interface Controller, and some say it stands for Programmable Interface Controller. The PIC microcontrollers are favourites of many hobbyists and professionals, because of following reasons:

1.Availability: PIC microcontrollers are very easily available in every electronics markets as well as online. One of the important reasons why they were a hit with hobbyists was that earlier Microchip used to provide free samples of PIC microcontrollers. So people who were starting with microcontrollers used to order them and once they learnt using it, they used to stick with it. Sadly, now Microchip asks you to pay shipping charges to India, which is higher as compared to the cost of PICs you can buy from the Indian markets.

2.Extensive Product Range: The PICs are available in numerous models and with different features, to suit the requirement of even the most demanding circuit designer.

3.Support: Since a lot of people use PICs, so a lot of online guides, forums, webpages and books are available on projects based on PICs. Also, the datasheets provided by Microchip for each PIC is very good and exhaustive.

4.Price: The PIC microcontrollers are relatively cheap and a basic PIC like PIC16F84, which would suit our application, would cost around ₹100.

A lot of people say that the architecture of certain PICs is tedious and is not helpful while programming in assembly language, but we need not worry about it since we will be programming in C++ language.

### 5.1.1 PIC: Internal structure

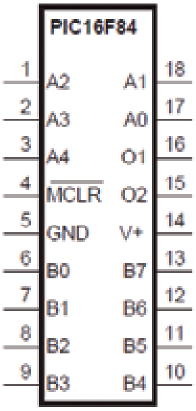
A PIC is a complete computer in itself. In the figure given alongside, a PIC16F84 microcontroller is shown. Any PIC would have three basic parts:

1.Central Processing Unit (CPU)

2.Program Memory (PROM)

3.Input Output Ports (I/O Ports)

The CPU is the 'brain' of the microcontroller. It reads the instructions from the microcontroller's memory and executes them. While doing that, it can store data temporarily in RAM (Random Access Memory) and retrieve it. The general purpose working RAM in a PIC is also known as "file registers."



Pin diagram of PIC16F84

The program memory of the 16F84 is of flash type which means that it can be programmed (stored with data) and erased electrically. So, you can store data in your PIC 16F84, erase it and then restore the data any number of times you want. Also, the memory is non-volatile, which means it retains its contents even when powered off.

Every processing unit has number of ports, which essentially is a group of terminals (or pins) used to transfer data. Generally, a port has eight pins. The PIC16F8X has two ports, PORTA and PORTB. Now, these ports are Input / Output Ports, which means they can work as both the input and the output for the data. Moreover, each pin is independent and you can even make two different pins of a same port do different tasks that is one working as input and the other as output. Every port has a register associated with it called a TRIS register. The purpose of TRIS register is to define which pin of that particular port will work as an input and which will work as the output. The TRIS register of PORTA is called TRISA and TRIS register of PORTB is called TRISB. A TRIS register consists of as many bits as there are Input/ Output pins in that port. Assigning the value '1' to a TRISA bit will make the corresponding pin work as an input, while assigning vale '0' to the bit will make the corresponding pin work as an output.

Each output pin of 16F84 can source or sink 20 mA as long as only one pin is doing so at a particular time. For more information on PIC 16F84, you can refer to its datasheet.

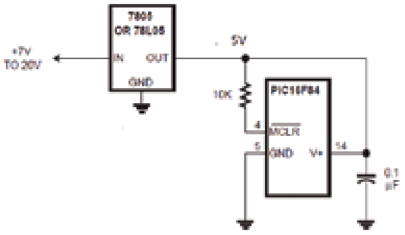
### 5.1.2 Power requirements

According to datasheet of 16F84, it requires 5V power supply, but it can practically work on any voltage between 4.5V to 6V.

Although the PIC consumes only 1 mA or even less at low clock speeds – but the power supply must also provide the current flowing through LEDs or other high-current devices that the PIC may be driving.

The circuit also has a 0.1-μF capacitor connected from pin 14 to ground, mounted close to the PIC, to protect the PIC and other components from electrical noise. This capacitor should always be used, because you can't be sure that the power supply available to you doesn't have any AC component present in it.

The MCLR pin, which stands for Master CLeaR, is a pin which is used to 'reset' the chip, which means clearing the data registers of PIC. This pin is active low, which means that it gets active and performs its function when provided with low voltage (OV), and is passive when supplied with high voltage (5V). Normally it is connected to 5V through a 10k resistor.



Grounding MCLR will clear RAM and reset the PIC. It is always a good practice to ground the MCLR momentarily as you switch on your circuit, so as to be sure that the circuit doesn't have any 'memory' of its previous operation.

## Clock requirements

All CPUs need a clock so that they can time their instructions. For example, if you want a particular instruction to be executed 5 seconds after the last instruction is executed, you have to provide some timing device which the processor will use to know when those 5 seconds get over. This timing device is called 'clock' and its frequency is called 'clock frequency'.

The higher the clock frequency, higher is the speed of processor. Generally, all processors have a maximum clock frequency which can be provided to them, in case of 16F84 it is 4MHz. Also, there is no lower limit to the clock frequency. The lower clock you provide, slower will be the processor, and lesser power it will consume.

There are two commonly used methods to provide clock:

1.Using a resistor and a capacitor: It makes use of inexpensive capacitor and resistor, but the clock may sometimes not be accurate, so you should avoid using this method when you need very accurate clock.

2.Using Oscillator crystal: A crystal is another electronic component, which is cheap and provides very accurate clock. We would be using this method for our robot.

## 6 Designing the circuit

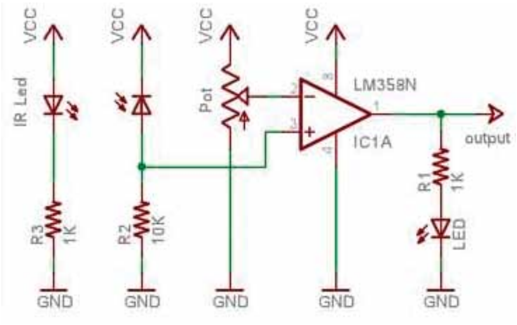
Now that we have decided that we will be using a pic microcontroller and two IR sensors, in this section we will understand the circuit.

### Circuit for the sensor

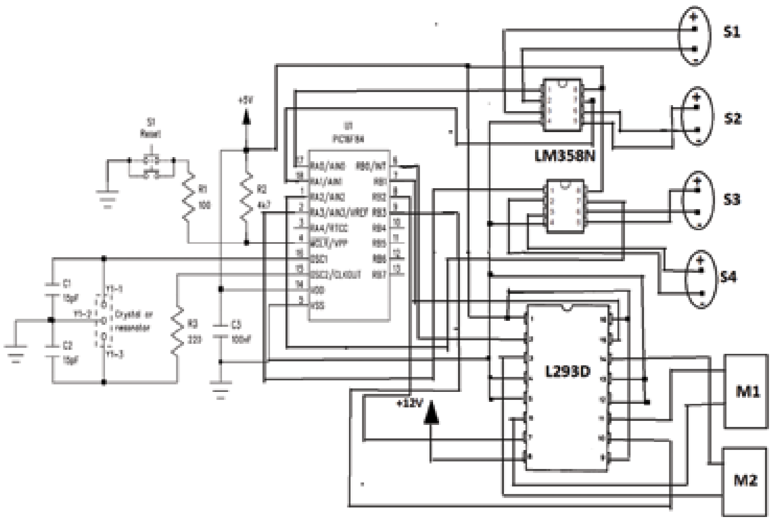
The IR sensor will require one IR transmitter and one IR receiver. The IR transmitter will target its radiations on the surface below it. When there is white surface below the sensor, it would reflect most of the radiations incident on it (because white colour is almost a perfect reflector). Some of these radiations would reach the IR receiver and the IR receiver would then start conducting. In that case, the terminal two of the Opamp would be grounded. Terminal three is grounded throughout. So, the potential difference between the terminal two and terminal three would be very near to zero. The Opamp output terminal would thus give zero voltage as its output. This output is logic zero. But when the surface below the sensor is black, the radiations emitted by IR transmitter would not be reflected back by the black surface (black colour is almost a perfect absorber). So, no radiations are received by IR receiver and it offers open circuit. In that case, terminal two will remain connected to high voltage, and the difference between terminal two and terminal three (which is grounded) would be maximum (nearly 3 Volts). This difference is amplified by the Opamp and the output would then give around 5 volt at terminal one. This output is logic one.

Note that a LED is also used in the circuit for sensor. It is just to check that whether sensor is responding to the black and white surfaces. The LED should not glow when there is white surface detection (logic zero, thus no voltage at output of the Opamp), and it should glow when the sensor detects black surface(logic one, thus 5V at output of Opamp.)

Now, let us study the complete circuit. The pic microcontroller has been provided with a clock of 4 MHz with the help of crystal oscillator. This method is already discussed before.



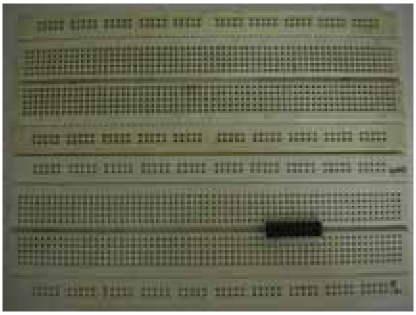
We are going to use PORT A of PIC 16F84 as input to sensors and PORT B as output to motors. Though we are going to use two sensors, but in the following schematics I have given the wiring for four sensors, to leave scope for improvement. Once you have successfully made a line follower, you can make a more advanced line follower which can respond to intersections and acute angles, and for that purpose you would



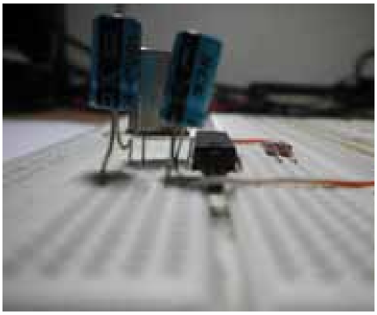
need more than two sensors. So it's better to include four sensors now only, so that you don't need to make the circuit again when you decide on making the advanced line follower. However for now, we will only use S1 and S2 as left sensor and right sensor respectively. S1 will give input to pin AO (called RAO) and S2 will give input to pin A1 (called RA1). The output to the left motor is given by pins RB3 and RB2, and the output to right motor is given by pins RB1 and RBO. Both these outputs are first fed through L293D motor driver, which provides necessary amplification required to driver a 12V DC motor.

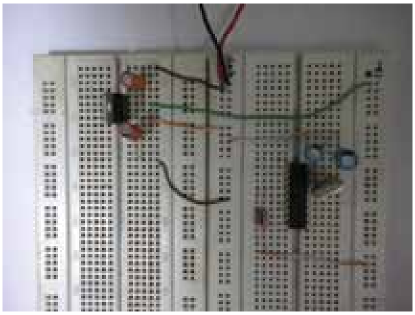
## 7 Making the Circuit

We would be building the circuit on breadboard. It would be good if we make the circuit on two breadboards joined together so that we may have enough space for wiring every component.



Two breadboards joined together and a PIC16F84 placed on them is shown in the following figure:





The next step is to make the clock circuit for the PIC using a 4 MHz oscillator. It is shown below:

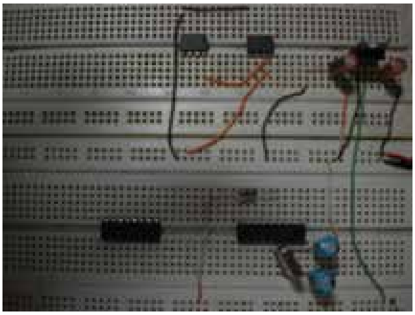
The next step would be to make the voltage regulator circuit to convert 12 volt input from battery to 5V. This circuit is built alongside the PIC Circuit, and the PIC is powered using this circuit.

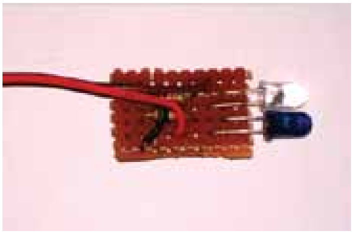
In the next step the motor driver IC L293D and the Opamps LM358N are placed.Also Opamp is powered.

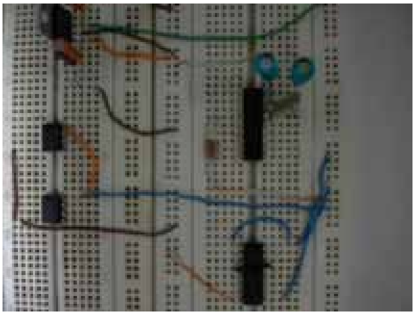
Finally all connections except those for sensors and motors are made and we get the following circuit:

The connections for sensors and motors will be done after the breadboard is mounted on robot body.

For making the sensor, you will have to do soldering because breadboards are too large for a small sensor and it is difficult to place a breadboard at the position where the sensor is usually







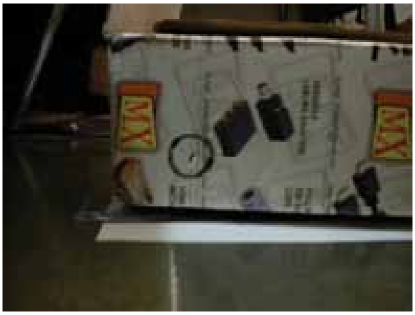
mounted. So, if you don't know how to solder, you can take help of someone who knows it. Following is the snapshot of the sensor I made on a PCB based on the same circuit diagram for a sensor given earlier in chapter 'Designing the circuit'.

## 8 Building the body

We would be following the differential drive type locomotion technique as it is easy to implement and very efficient as well. Also, you don't need to go for wood to build a body for the line follower since you do not require your robot to have that much your strength. The best material to make the body for your first robot is the cardboard. Take any cardboard box which you may have at your home. It will already be in shape and you would be spared with the trouble of designing a body, fitting various parts of the structure together. A cardboard box of right size would be perfect for application. The right size would be according to the size of your motor. Ideally, it should be as wide as twice the length of your motor and its length should be such that it could accommodate the breadboard on its top. I found one such box and I believe it is just perfect for our application.

After selecting the material for body, we need to decide which motor we should use. The DC geared motors provide good torque, so we would be using them. Also, we don't need too much speed in locomotion. Too much speed would require a very fast processor and efficient code since if the speed is high, the processing unit will have lesser time to judge a turn and respond. So, a DC geared motor with the speed of 50-100 rpm would be ideal for us.

The next step is to make holes on the sides such from where the motor's axle would come out. The diameter of this whole should be equal to the width of the axle of your motor. Following figure shows the circular area which is to be removed for making the hole.

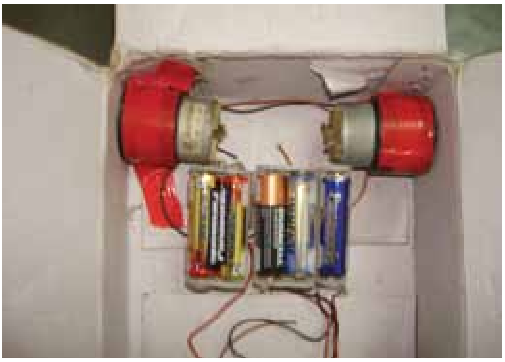


Note that the height of the hole from the base would be equal to the height of the axle of the motor from the ground when the motor is placed on ground in such a way that its curved surface touches the ground. After removing the highlighted portion of the box, the hole will be created. Make a similar hole on the other side also.

Now, keep the motors in place and insert their axle in the holes. Secure the motors in place with the help of some insulation tape.

Now, as the motors are secured in place, put and the wheels on axle and fasten them there with the help of glue. You can get these wheels from the toy shops. Also, you can remove the wheels from some old toy and use them over here. If the slot of the wheel is wider than the axle, you can roll some tape or paper on the axle and make it thicker and then apply glue so that wheels get fixed over the axle firmly. You need to search for other pair of wheels if the slot in the wheels of axle is narrow and can't accommodate the axle of your motor. Here is how it would look like after putting the wheels in place.

The diameter of the wheels should be such that there is some clearance of the body from ground. So make sure that the wheels are big enough so that we have at least ground clearance of 1 cm.



If you want, you can have a third wheel (pivot wheel) at the back. It would not matter much even if you don't have that. The body would just drag at that point and because our material is cardboard, the friction offered would not be too much.

# 9 Writing the code and Programming

After making the circuit and body, the next step is to write the code for the microcontroller. Since we have had a chapter on programming using C++ language, the code writing would not be difficult now.

## 9.1 Writing Code

A sample code for the line follower is as follows:

```
  #define le pin_a0    //defining pin a0 to work as
input for left sensor, s1
  #define re pin_a1    // defining pin a1 to work as
input for left sensor, s2
  #define lm pin_a2    //s3not used at the moment, can
be used for improving efficiency later on
  #define rm pin_a3    //s4, not used at the moment, can
be used for improving efficiency later on
  #define FWD 0x05// 01 01 both motors in forward direction
  #define RIGHT 0x04  // 01 00 left motor forward and
right motor stops
  #define LEFT 0x01   // 00 01 left motor stops and
right motor forward
  #define REV 0x0A // 10 10 both motors in rev direction
  void main()
  {SET_TRIS_A( 0xFF );          // FF=1111 1111, which
means we declare PORT A to work as input
  SET_TRIS_B( 0x00 );          // 00 = 0000 0000, which
means we declare PORT B to work as output
  while(1) //infinite loop, so that microcontroller keeps
on checking conditions repeatedly
  {
    if(le==0 && re==0)   // both sensors detect white
surface, so straight path ahead
      {
    output_b( FWD); // straight path ahead, so move forward
      }
   if(le==1 && re==0)   // left sensor detects black,
right sensor detects white, so left turn ahead
```

```
      {
      output_b( LEFT);     //  Turn Left
      }
       if(le==0 && re==1)  // left sensor detects white,
right sensor detects black, so right turn ahead
      {
      output_b( RIGHT);    //turn right
      }
       if(le==1 && re==1)    // Both sensors detect
black, the robot is perpendicular to the black line
      {
      output_b( REV);      //  Go back
      }
   }
   }
```

Every line of code is explained alongside in the form of comment. You can have more sensors and more cases to make your code efficient. But to start with programming, the code above would be sufficient.

Probably, coding is the most important part of making robot important because once you have made your robot, you can't just keep on changing the circuit or structure of the robot, but you can change the program as many times you want. So, once the robot is constructed, the only way to improve its efficiency is to improvise on the code.

### 9.1.1 Programming

Having written the code, the next step is to feed this code into the microcontroller. For this purpose you require two things:

1.Hardware Interface

2.Software Interface

We have written our code in C++, but the problem that microcontroller understands only binary language still remains. So we need software which converts this code into binary language. Such software is called a 'compiler'.

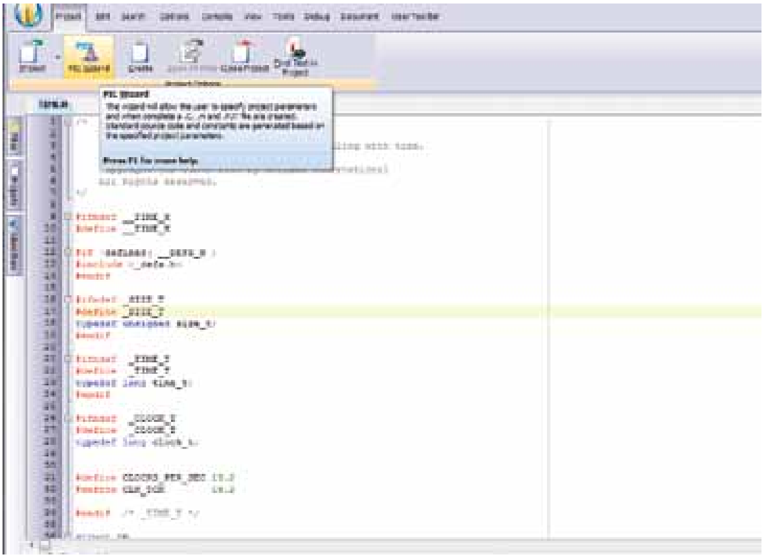
Now, the choice for your software interface depends on what type of hardware interface you choose. A hardware interface is essentially a 'programmer' which connects from one end to the port of your computer, and is connected to the microcontroller at the other end. It is essentially a physical channel through which your code 'flows' from your computer to the memory of the microcontroller.

There are a variety of programmers available in the market, and the cheapest one which can work with a USB port is available for around Rs. 2000. But, if you plan to spend that much, we would seriously advise you to buy PICKit 2 programmer, which costs around Rs. 2700 and is designed and supplied by Microchip itself, and is very user friendly. Mind it, it's a universal programmer and can program almost all the PIC's available in the market these days. If you want to continue working in the field of robotics and electronics, and want to stick with PIC microcontrollers, you should seriously consider buying it.

The PICKit2 programmer

### Programming using PICKit2

Following is a pic of PICKit2 programmer which you will have to procure from Microchip.

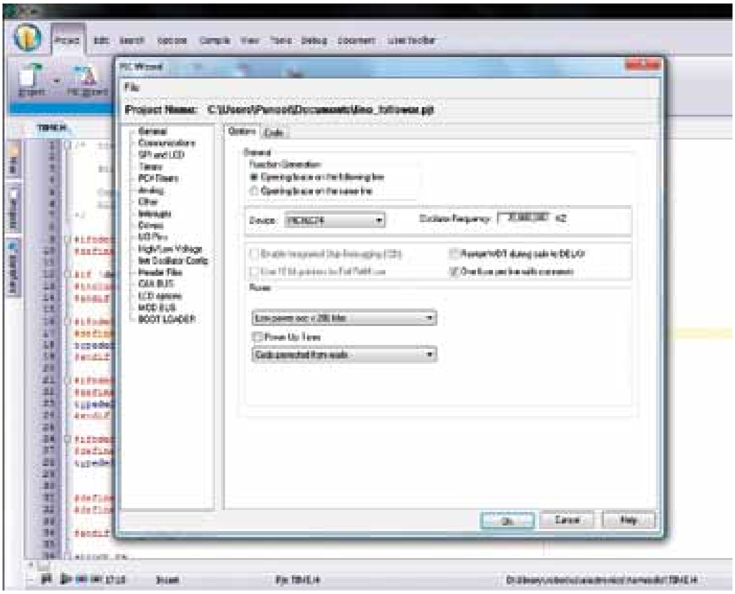

PIC Wizard

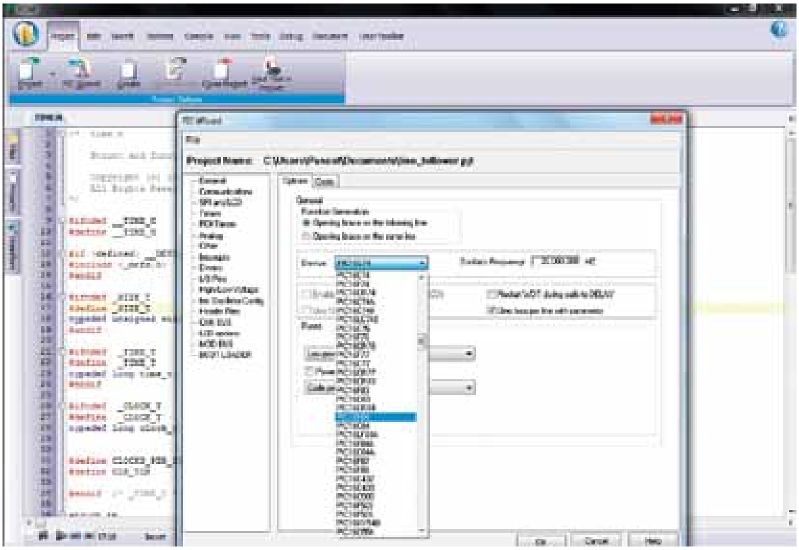
[illegible]

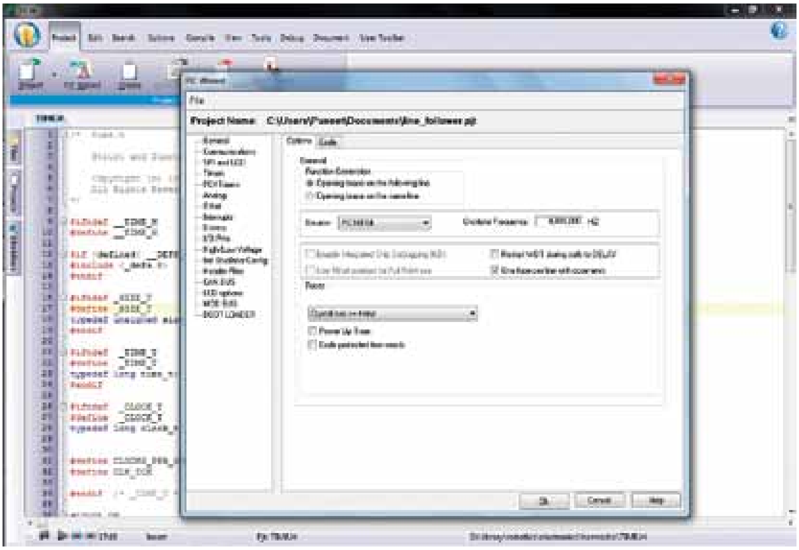
Press F1 for more help.

[illegible]

Go to Project Tab > Pic Wizard


Select the device from the drop down menu and enter the oscillator frequency


You can select from a wide range of PIC microcontrollers



Select crystal osc <= 4 mhz

We need a compiler to convert C++ code  to binary language. For this, we have software called 'MPLAB' from Microchip which is very good. However, for beginners, it is slightly difficult to learn. So we would use a compiler CCS PCW C compiler. We have had already discussed C++ programming on this compiling platform in the chapter 'Introduction to Programming'. This compiler is very easy to use and is available online. Following are the steps to be followed once you have installed this compiler and have the code ready.

The project is created and the screen where you can write code pops up, with all necessary libraries attached on its own. Write the code in the space provided and click on build all option under compile tab.

Similarly, follow the procedure to write the hex file to the PIC. The only difference is that instead of using the compile option, go to `File Tab > Import Hex`.

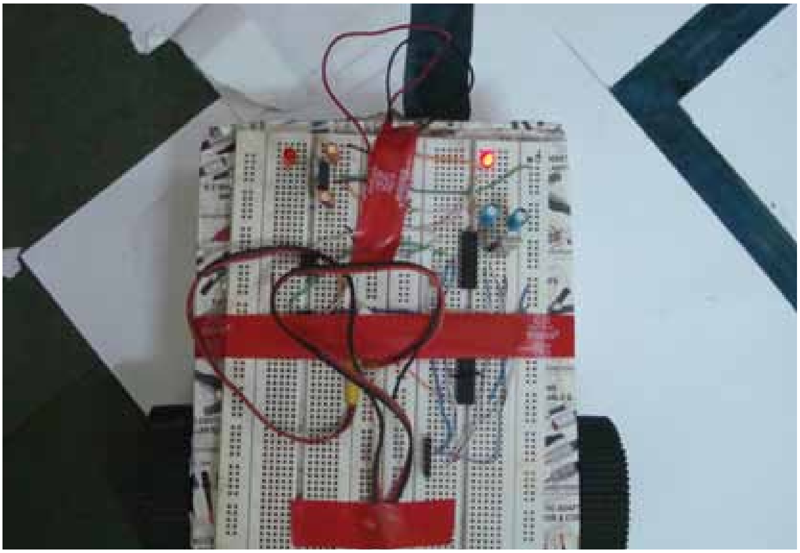
In the `Import Hex File` window that pops up, select the location where you have your hex file. Select the hex file and then click 'Write'.  The code is burned on the PIC.

## 10 Assembly

After you have made the circuit, built the body and programmed the PIC; you need to assemble everything to see if it works.

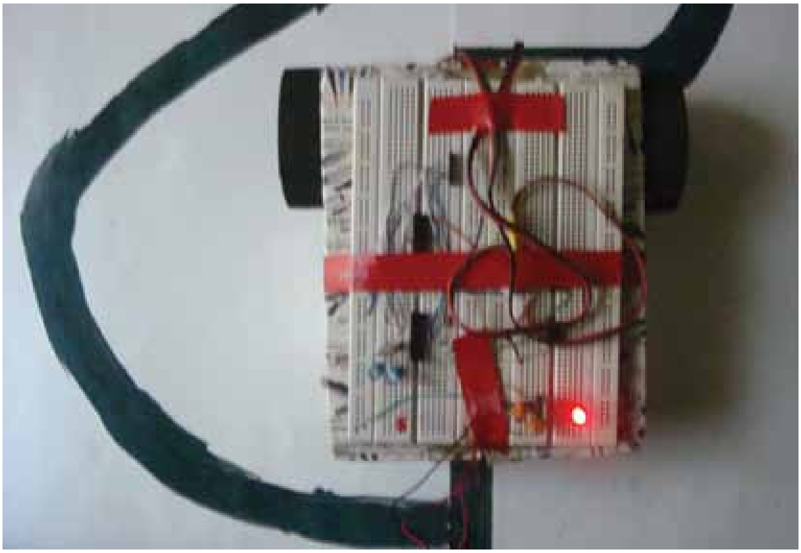
Provide voltage of 12V by using 8 X 1.5 V cells in series or a 12 V battery itself. Next you need to place the batteries in the body of the robot. Try placing it at the centre of the robot.

The sensors are mounted at front of the robot such that they are just outside the edges of black line on both sides. You can use either glue or tape



to stick them there.

The breadboard is placed on the top of the body and is fastened with the help of some insulation tape.



As soon as you power the circuit, the line follower should start working and should be following the line. Note that two LEDs have been added in parallel to output of the Opamps. They glow when any sensor detects the black line. When the left sensor detects black surface under it, the left LED glows to indicate that there is a left turn ahead. The robot turns left.

Similarly, when the right sensor detects black colour, right LED glows to show that there is right turn ahead. The robot then turns right.

None of the LEDs glow when there is straight line ahead and the robot keeps on moving straight. We built this robot on PCB also and here is what the circuit will look like when you solder it on a PCB.

Here, we used PIC 18F4220 instead of PIC16F84. PIC 18F4220 is more advanced than PIC16F84 and we used it so that we have some scope for better algorithm and logic, since we plan to make a more efficient line follower in future.

We also built the body using wood instead of cardboard and used four motors instead of two so that it could have more power and ability to climb inclined surfaces too.

**Video!**

On this month's DVD, we've included an exciting video of our robot in the making. You'll find the step-by-step instructions easy to follow, especially if you are not familiar with electronics and programming. Write to ***editor@thinkdigit.com*** and tell us about your experience.